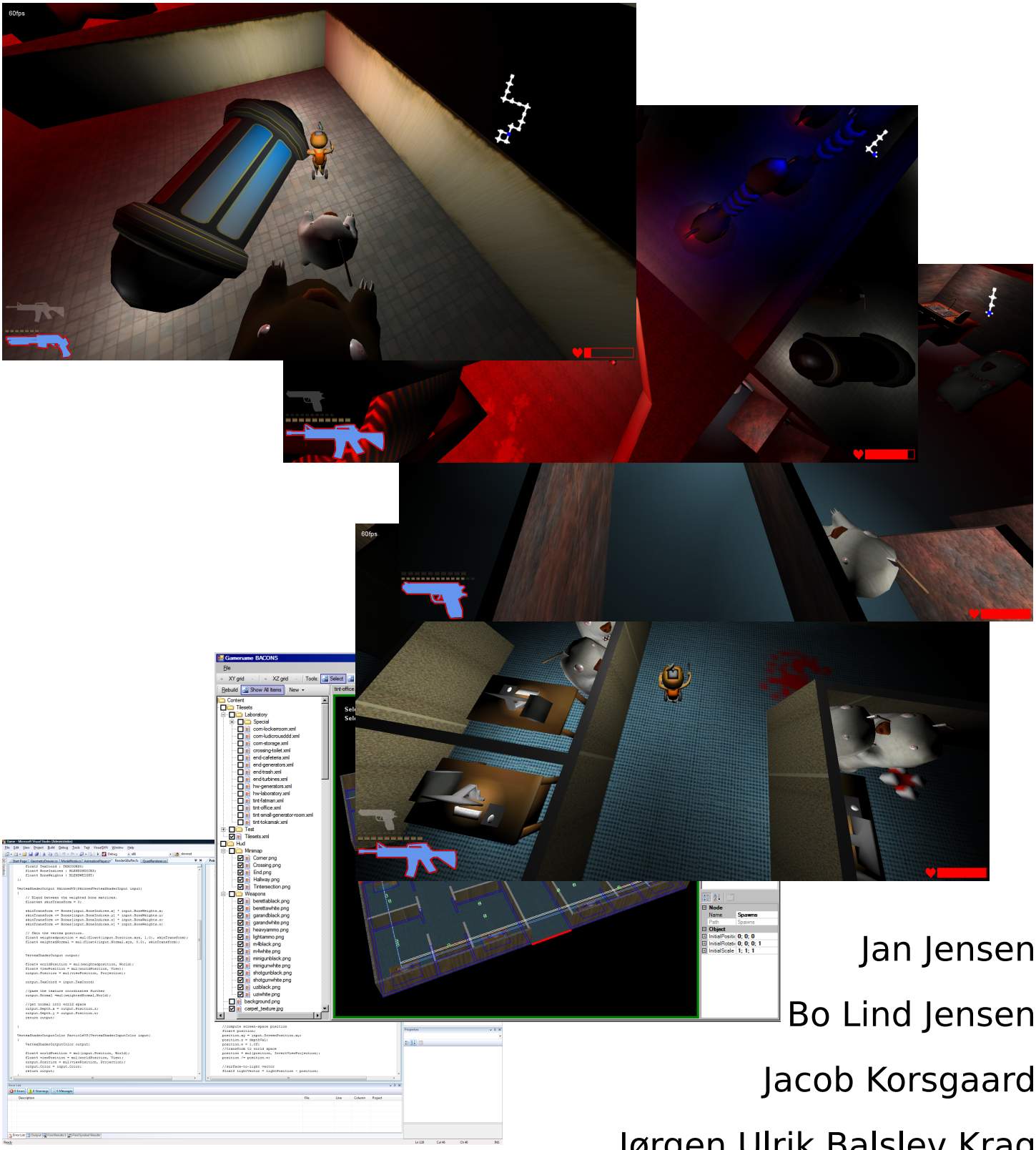
ORZAM

Oblivious Rampaging Zombies And Monkeys

The **EGGS** Engine | The **BACON** Editor | The **ORZAM** Game



Jan Jensen

Bo Lind Jensen

Jacob Korsgaard

Jørgen Ulrik Balslev Krag

Dan Leinir Turthra Jensen

Fall semester 2008
Game and Engine Programming
Aalborg University

AALBORG UNIVERSITET

Department of Computer Science
Selma Lagerløfs Vej 300
DK-9220 Aalborg Ø
Telephone (+45) 9635 8080
http://www.cs.aau.dk

Title:
Creating The ORZAM Game Using EGGS and BACON

Project theme:
Game Development and Game Engine Programming

Project period:
Fall Semester 2008,
2th September to 19th December.

Project group:
sp1e082

Participants:
Jan Jensen
Jacob Korsgaard
Jørgen Ulrik Balslev Krag
Bo Lind Jensen
Dan Jensen

Supervisor:
Bjørn Haagensen

Print run:
7

Pages:
105

Appendices (number, type):
1 CD-ROM with source and binaries

Abstract:

To learn about game engines, we designed a game and built an engine and level editor for that type of game, using C# and the XNA Framework. The engine has 2D physics and a 3D graphics renderer based on Deferred Shading. A randomizing level generator was implemented which is based on a tile engine. Tiles for a game are authored in the editor and are saved using an XML based scene graph system.
The result is the ORZAM game, a fully functional multi directional shooter pitting a monkey against a horde of zombie bears, which was implemented using the game engine and editor.
The engine and editor are functional and fully reusable for new game projects, although specialized in 2D games with 3D graphics.

This report deals with the development of a game engine, an editor and a game prototype developed using the engine and editor.

We assume the reader has basic knowledge of computer science and an interest in game development. For the technical parts, a preexisting knowledge of Linear Algebra is assumed.

Citations in the report are written as [1], where the 1 corresponds to an entry in the bibliography, which is located at the end of the report. **Bold** and *italic* are used to emphasize words, no specific convention is applied.

A CD-ROM is attached to this report, it contains the source of the implemented software.

Jan Jensen

Jacob Korsgaard

Jørgen Ulrik Balslev Krag

Bo Lind Jensen

Dan Jensen

CHAPTER
1

Introduction

Given the overall semester theme "*Game Development and Game Engine Programming*", we are presented with the task of implementing a game using a game engine.

The purpose is to give us detailed knowledge and an understanding of how a game engine works, as well as give us knowledge on using a game engine in practice and to account for its strengths and weaknesses.

To aid the project we are taught three courses, "Game Programming" "Computer Graphics Programming" and "Animation Techniques", which we intend to benefit from in the process.

Based on the purpose of the semester, we formulate the following goal for the project, in order to limit its scope:

> *We want to get hands-on experience with game engines by implementing a game using a game engine.*

We base our project on the goal mentioned above and begin by producing a suitable game design, which will serve as case in the project. The project focus is on game engines, therefore the game design phase will be brief. The game design case will be used to specify the requirements for a suitable game engine and for implementing a game.

1.1 Report Structure

Chapter 2 on the facing page contains our game design case on which we base our project.

Chapter 3 on page 12 presents game engines and the motivations for using one. It will also describe some typical features of a game engine.

Chapter 4 on page 26 considers whether to build an engine from scratch or use an existing game engine. A choice is made to build our own game engine.

Chapter 5 on page 31 documents the implementation of our own game engine.

Chapter 6 on page 59 describes our implemented level editor, which is a part of our game engine.

Chapter 7 on page 65 contains our game implementation. The initial game design is implemented using our developed game engine and level editor.

Chapter 8 on page 89 will contain three sections: An evaluation of our engine, a conclusion on whether we have reached our project goal and finally a section on future work.

CHAPTER

2

Game Design Document

This chapter contains the design document for our game. The game design document is separated into three parts:

Key Concepts: These are the major decisions made on what kind of game we want to make.

The Setting: Describes the background story, characters, atmosphere and goal of the game.

Gameplay: A more in-depth description of the gameplay and game mechanics we want to implement.

The actual implementation of the game can be found in chapter 7 on page 65.

2.1 Key Concepts

As mentioned in the introduction our project focus is on the more technical aspects of game programming. This means that game design is not a crucial part of the project.

As a result, we want a simple and proven game design, which enables us to focus on technical aspects and not on designing games. We want things to be simple, but we would also like to work with advanced computer graphics, in order to utilize the Computer Graphics Programming Course. Furthermore, the design should be advanced enough that we encounter sufficient challenges in using game engines.

Genre We choose to create a Multi-Directional Shoot 'Em Up game, where the player has the freedom to move and shoot in any two-dimensional direction, with the camera positioned above the action following the player around. Examples of games that uses this method are "Robotron: 2084" and "Geometry Wars".

We personally find this type of game fun and entertaining. The gameplay seems simple enough that we can create a working prototype within the project period.

Atmosphere and Game World We want to make the game fun and a little weird. Instead of creating a realistic game world, we create a fun and silly world, as we personally prefer this kind of game.

Game Extent As this is a semester project, we aim to implement a working prototype of a game. However we will still aim to provide a gaming experience of a reasonable extent, that is a game that takes more than a few minutes to complete.

Instead of creating a sufficiently large "world" with tons of content, we want to generate unique worlds using as little content as possible.

Players The game should be playable for a single player, so that you do not need to be multiple players to play the game. However the genre is well suited for multiple players, so we would like to implement some sort of multi-player functionality. As networking is beyond the scope of this project, players should play on the same machine.

Graphics The game should be in 3D for us to make use of the Computer Graphics Programming Course. Also this makes the technical challenges more interesting.

2.2 The Setting

In the previous section it was stated that the design of the game should be fun and if possible a little bit weird. This means that we need a game world where the characters should not take themselves too seriously, and neither should the player. So if we can come up with an original concept with original characters, we have the freedom to make them as weird and silly as we desire. To enforce this silliness we will need sound effects which will go with the overall theme.

For the setting of the game, we will let it take place in a mad scientist's underground laboratory. This way we can restrict the player movements but still make the game world open enough that the player has some choice when moving around. A lab will have lots of rooms of different kinds, ranging from a cafeteria, where the loyal henchmen can take a break, to weapons storage where the scientist stockpiles his weapons of mass destruction.

One character you might find in a scientist's underground laboratory is a lab monkey. Thus we have the hero character of the game. But a monkey on its own is not weird enough, so the monkey could have some extra gadgets attached to it, which might guide it or let us make a story about the player controlling the monkey through this device. Thus we end up with a monkey character with a radar on its head. To give the monkey some reason to get out of the lab, something has to have gone awfully wrong. To provide this we introduce a nemesis. For this position the illogical choice would be zombies, as nothing could be more out of place in a top secure complex, except maybe for flowers. But zombies are not nearly fun or weird enough, so the zombie is made into a cuddly zombie bear, which looks harmless but in reality be a vicious killing machine.

With the characters and the environment in place, it would be a good idea to finalize a story that encompasses these traits. Such a story could then be expanded upon in the game as the player advances.

2.2.1 Background Story

For many years the monkey, our hero, had been held captive as test subject by an evil scientist. Over the years the monkey had been injected with various intelligence enhancing serums, but with each IQ point the monkey gained, it became only more neglectful towards its physical condition. As time passed by, the monkey became indifferent and too easily exhausted to walk. The scientist had put a portable radar on the monkeys head in an attempt to motivate the monkey to find the exit of the lair; the radar was designed to always be pointing towards the nearest exit. The monkey, however, had no interest in escaping - it had become too lazy to try. Thus the scientist, without planning on it, had created the cageless monkey prison.

Trying to get a lazy monkey to escape was not the only thing occupying the scientist's mind. The scientist was madly fixated with preserving all kinds of wildlife, and consequently he started gathering bears from all over the planet. As it is common knowledge that zombies need no food to survive, and only occasionally ate brains, it was considered to be the best option to turn the animals into zombies rather than buy food for them. So the scientist injected all the bears with a serum which turned them into mindless, sometimes brain eating, zombies. Loosing a few workers during annual bear count was considered an acceptable risk, outweighing the cost of food numerous times.

As the scientist now had a great number bears, the safety risk became ever increasingly larger. This eventually led to the construction of the Defence-O-Magic system, a single machine intended to shoot and stop the bears. It used an unlimited supply of Bio-Goo to construct its specially tuned zombie stopping projectiles.

Disaster strikes...

One day all the doors on the holding pens slammed open and the bears started pouring out. Eventually a single bear reached a Defence-O-Magic cannon and got its zombie DNA mixed with the cannon's Bio-Goo supply, which had an unfortunate design flaw that a single cannon's supply was linked to all cannons in the lab. This new zombie Bio-Goo had the side effect of the cannons shooting infectious bears, instead of the intended zombie stopping projectiles. This new race of zombies was created with a higher than normal metabolism, making them grow to an unnatural size in less then a minute. It was only a matter of time before a rambling zombie

accidentally ran into the magnetic field of the core reactor, thus shutting it down leaving only the emergency power on. The monkey, almost hypnotized by the headlights on a nearby Segway, was scared of the different growling noises the bears made in the dark. It knew that given its cardio and the now pitch darkness of the laboratory it would never make it out alive. So it grabbed the Segway and scooted off for the elevator, and towards freedom.

To be continued in the game...

2.3 Gameplay

The overall **goal** of the game is to have the player reach the exit, thus advancing to the next level and eventually winning the game by reaching the exit on the final level. To make this task a bit more difficult, the player's path should be hindered by an opposing force.

One type of **enemy** would be a slow moving enemy character that attempts to catch the player if he gets within line of sight. These enemies will hurt the player if ever allowed too close. A problem which occurs with this type of enemy is the player being able to simply run away, without ever being in danger or having to shoot.

So another kind of enemy is introduced. This enemy will **shoot** at the player, so the player has to change direction, or at some point stop, to avoid damage. A simple cannon shooting fireballs is not funny, so instead this cannon is set to shoot morphing enemies. The projectile starts out as a small zombie which, if left alone, over time grows larger and faster. Eventually the player becomes unable outrun them, and will have to shoot them to survive.

To enforce the feeling that there are multiple types of enemies, each of these could be given a unique sound effect.

To have zombies waiting for the player, a mechanism that allows for zombie respawns will be implemented. Although we never want the player to run out of zombies to shoot, there should be periods where there are no enemies. This gives the player time to mentally recuperate, while looking at his ammo and health status. Also, this downtime will make the next encounter with a lot of zombies more thrilling, instead of constantly being under attack and mentally alert.

In case the players' health reaches zero, the player will die and the level will be reset.

The player will need to be armed with some kind of weapons to defend themselves. At the start of the level the player has a low powered weapon, which is useful until a better weapon is found. If a player ever runs out of ammunition, he will have to find some more or have his brains eaten.

Since the player can run out of ammo or health, some kind of ammo and health power-ups are needed. The power-ups are items on the floor the player can pickup to replenish ammo and health. Powerful zombies could carry more powerful weapons, giving the player a reason to let them grow. This puts the player in the situation where they have to choose whether it is worth letting a little harmless zombie grow into a vicious killing machine, to get a chance at a better weapon, or kill it immediately while it is relatively harmless.

To differentiate between powerful and weak weapons, two different types of ammo are used. The powerful weapons and their ammo should be scarce in the environment but droppable by the powerful zombies and the zombie cannon.

Each time the player plays the game, the levels should be different. But as we do not want to create a lot of different levels by hand, new levels are to be procedurally generated.

When multiple players play the game, friendly fire will be activated, so the players have to be more cautious when shooting. If one player dies, all players lose the game. This is done to force the players to think as a team.

2.4 Design Summary

The design calls for a wacky multi-directional Shoot 'Em Up game, in which the levels are generated on the fly using as little manually authored content as possible.

The action in the game takes place in 2D, but is rendered in 3D; We call this a 2.5D game.

The game will feature both single and multiplayer gameplay, taking place on the same machine.

CHAPTER

3

Game Engines

This chapter presents game engines, our understanding of what a game engine is and the motivations for using one. We will be describing the typical features of a game engine and how we expect to use one for the development of our game.

After describing game engines in general we proceed to detail the features we intend to use from a game engine.

3.1 Game Engine Fundamentals

Josh Petrie defines a game engine as follows:

> An "engine" is a collection of robust, reusable software subsystems (possibly including both code libraries and tools) designed to facilitate the development of actual games by addressing specific requirements.

We agree with this definition, except we do not consider tools a part of the engine. Instead these are a part of a general Software Developers Kit (SDK). We believe these are separate because the engine can be used without the SDK, where the SDK is only useful when accompanying the engine it was designed for. Furthermore, when the game is running it is using the game engine, but not the SDK.

We believe that a game engine in its essence is a code library, meaning that it is a collection of generic classes which are used when running a game. Game engines vary greatly in complexity. Some provide only a few helper classes, while others are complete systems that allow developers to create games visually using drag-and-drop.

3.1.1 Motivation

The main reason for creating and using game engines is the same as creating and using code libraries in general, which is reusing code.

Engines are also often accompanied by an SDK, which has the benefit of giving content creators something to do early in the development process. They can begin using the tools right away and expect the resulting content to be playable using the engine.

3.1.2 Typical Features

A game engine will typically provide one or more of the following features:

Content Management: Engines often have functionality for importing different types of content such as 3D models, textures, sounds, scripts and levels. An engine's SDK often define which content file formats can be used.

Level Representation: The objects of a game world are usually represented in the game using a level data structure, defined by the engine. Typically these are a form of tree, and are called Scene Graphs.

Rendering: Some engines have a rendering subsystem for drawing the game world on screen. These systems are often tuned to the type of game the engine is designed for. Some are very advanced providing features such as animation of 3D models, advanced lighting and special effects.

Platform Independence: Engines will often abstract the platform used away from the developers. This way a game will only have to be developed for the engine to be able to run on all platforms supported by the engine.

Input: Engines can also abstract away the specifics of input handling.

Physics: Engines often have physics systems that define how objects interact with the world and each other. The functionality ranges from full-fledged real-world physics simulations to simple 2D collision detection.

Artificial Intelligence: Techniques used to create the illusion of intelligence for the non-player characters in the game.

Networking: Systems for handling network communication. Often used for multiplayer games.

Game-Loop: Some engines provide a game-loop. During the game-loop all systems of the game are updated. Typically, player input is read, the game world is updated and drawn to the screen. The game-loop repeats itself until the game has ended.

Game State Management: A game often has several different states such as a menu, being paused, searching for network games and of course being in-game. Engines often have functionality for switching game states and displaying menus.

Storage Management: Saving and loading game progress is also often part of an engine's responsibility. This is even more important if the game is supposed to run on several different platforms.

Implementing all the above features and getting them to work together properly is a very expensive and time consuming process. Therefore middleware systems are often incorporated as subsystems in the engine. Examples are the Havok Physics[1] engine, NaturalMotion Euphoria[2] engine for procedural animations and fmod[3] interactive audio engine.

When an engine has reached release stage, most game developing companies reuse the same game engine to create multiple games, sometimes modifying the engine to make it handle specific cases.

3.1.3 Specialization

Most game engines are tailored to a specific genre of games, meaning that engines for different genres generally handle the same concept, such as network and collision code, but in differently.

The physics system in a real-time strategy engine like "RNA"[4], which was used to create "CnC Red Alert 3"[5] might be less accurate, than the one used in an FPS game engine such as "UnrealEngine3"[6].

Although the rule is that game engines are normally tailored to a specific genre, there are exceptions to this rule as they can still be applied to other game genres. An example of this is "UnrealEngine3", which has been licensed to be used in numerous upcoming Massive Multiplier Online Games like Mortal Online[7] and Huxley[8].

3.1.4 How our game would use a game engine

We can now describe which engine subsystems we require for our game according to its design as seen in chapter 2 on page 9.

Basic 2D physics: Because the game action is in 2D we need a basic 2D physics system.

Content Management: The engine must be able to import 3D assets we create using our favorite 3D modeling packages. We also need support for other content types, like bitmaps, sound effects and music.

[1] http://www.havok.com/
[2] http://www.naturalmotion.com/
[3] http://www.fmod.org/
[4] RNA (not an acronym) is an updated version of the SAGE (Strategy Action Game Engine) used for CnC Generals, and The Battle for Middle-earth series
[5] http://www.commandandconquer.com/
[6] http://www.unrealtechnology.com/
[7] http://www.mortalonline.com/
[8] http://www.huxleygame.com/

Procedural Levels: The engine must be able to generate randomized levels in 2D.

Multiple input devices: To allow multiplayer gameplay on the same machine, the engine must handle input from several devices simultaneously. The engine should be able to handle gamepads and keyboard input.

3D Graphics: The engine must be able to render 3D graphics. It must also be able to handle animated 3D models.

Available source code: The most high-level parts of the engine source code should be available. We are not interested in the most low-level details, such as platform specific details on how storage management is implemented.

The following sections will elaborate on scene graph, 3D graphics, animation and physics subsystems we need.

3.1.5 Scene Graph

A scene graph is a way of representing data in a game world; as in a data structure where parent nodes affect the child nodes. The scene graph can be used to describe logical relationship between entities, and to increase performance. In essence a scene graph is an n-tree that can have as many children as is needed[2].

When describing a logical relationship between entities, the nodes involved have some dynamic spatial relevance in the game world. An example of this could be a car. A car is made up of a bunch of components, where the main component is the chassis, then some doors, windows, an engine, and eventually a chauffeur which could get in and out of the car. Making the car's chassis the parent node means that if the car chassis translates in a direction, the child nodes get the same action applied, meaning the same applies to rotation, etc.

Furthermore each of the child nodes can have individual actions applied to them; one such action could be damaging the engine or one of the windows of the car, without the chauffeur taking any damage.

Overall there is no clear definition on what a scene graph is, as programmers implement and adapt scene graphs to suit the needs at hand. There is a plethora of different nodes one can create and implement for a given graph. Terrain, character, camera, and particle system nodes are just some. The implementation of the scene graph depends on the requirements of the game[9].

3.2 Graphics Rendering

This section presents the basics of rendering 3D graphics. The pipeline we describe is divided into three stages: the application, geometry and rasterizer stages. An overview of the pipeline can be seen on figure 3.1 on the next page.

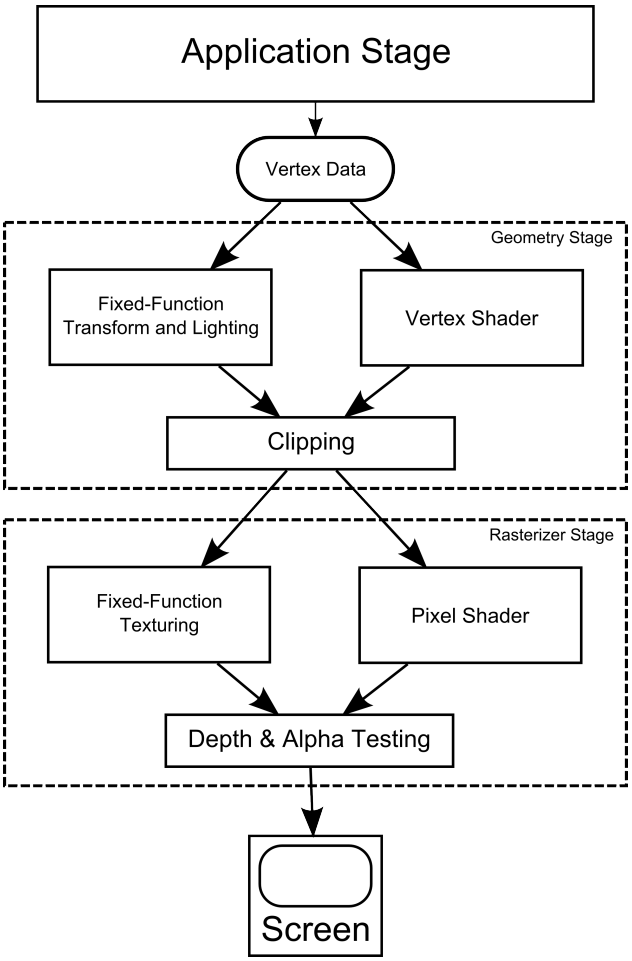


Figure 3.1: An overview of the rendering pipeline. The arrows indicate data flow.

3.2.1 The Application Stage

The application stage is the part of a 3D application or game that runs entirely in software on the CPU. Commonly this is where we have the game-loop with an update and draw function. In the update function we position all the pieces of the game like characters, environment, cameras and lights. In the application stage is where we would put physics, collision detection and basic game logic.

When the setup is complete we prepare the scene to be rendered by the graphics pipeline. The 3D geometry we want to render is defined using vertices, which are simply points in 3-dimensional space, although when represented in our game, we may want to attribute them extra information such as color, texturing coordinates and normals (we will get back to that). The vertices in a piece of 3D geometry (or model) are in what is called *object space*, which means they are relative to the origin of the object they are a part of. When we want to draw the model somewhere in our world, we first need to describe how to transform its vertices into *world space*, to make their coordinates relative to the world. This is done by creating a **World** transformation matrix we can multiply with the vertices to apply translation, rotation and scale to the object.

Next we want to describe where we want to see the world from. This is done using cameras. A camera is defined using a point to represent the position of the camera, another point representing the target of the camera, or what it looks at, and finally a vector representing which direction is up for the camera. With this information we can create a **View** matrix, which will move all our vertices into *camera/eye space*, where their positions are relative to the camera, with the negative Z-axis being the direction the camera is pointing, Y-axis pointing up and X-axis pointing to the right.

Finally, we want to describe how to position the geometry on the screen, that is moving the

vertices from *eye space* into *screen space*. We want to project the 3D geometry onto a 2D plane (representing our screen), by using a **Projection** matrix. In *screen space* the upper left corner of the screen is $(-1, -1)$, the lower right is $(1, 1)$ and the Z-axis is the depth ranging from -1 to 1. This is the unit cube called the canonical view volume.

The two most common projection matrices are perspective and orthographic. Perspective is the most commonly used as it mimics the way people see the world, specifically how objects appear smaller the further away they are from the camera, and parallel lines can converge on the horizon.

The main characteristics of a orthographic projection are that parallel lines remain parallel and objects maintain their size in distance. The difference between the two is illustrated in figure 3.2. Together the projection and view matrices form the view volume, or **frustum**, which contains the part of our world that will be rendered to the screen.

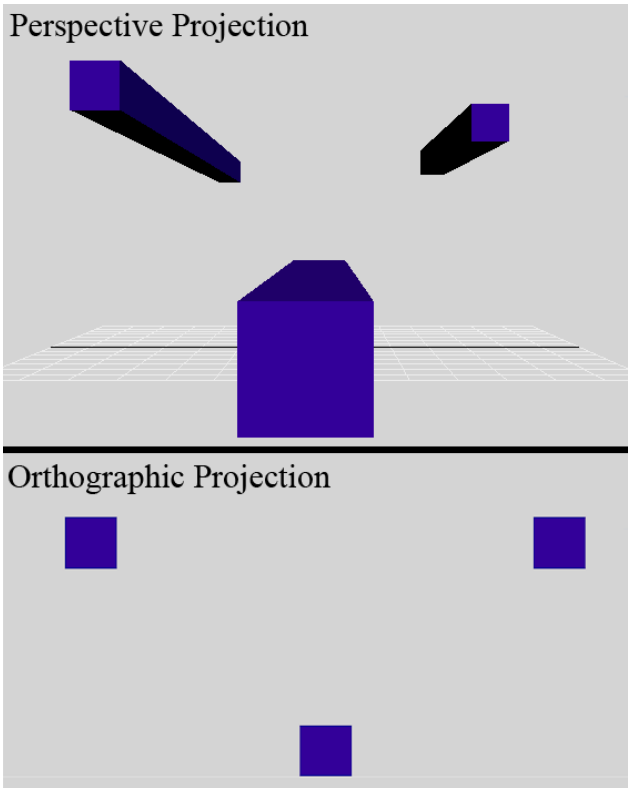


Figure 3.2: The same three elongated boxes viewed from the front using perspective and orthographic projection. Because lines remain parallel with orthographic projection, the three boxes appear to be equal size and at the same distance.

By now we are ready to pass the data to the graphics device. For each piece of geometry we need to pass:

- A **Declaration** of what information we have attached to the vertices, most commonly a position, a surface normal vector, a color and a set of texture coordinates.
- Our vertices put into a **VertexBuffer** which can be loaded onto graphics memory.
- The type of **Primitive** we want to draw: Points, lines or triangles. For specifics refer to Real-Time Rendering[1].
- Any light sources in our scene.
- Any **Textures** we want to apply to the geometry.
- And our three matrices: **World**, **View** and **Projection**

For more complex models we would also want to use an **IndexBuffer**, which is a list of indices of which vertices from the VertexBuffer to use. To illustrate the gain of using an IndexBuffer imagine a cube, which we would need 12 triangles to draw, two for each face (see figure 3.3). When using the **TriangleList** primitive type, we need 3 vertices for every triangle, that makes **36** vertices for a simple cube, and as we have seen each vertex can have a large amount of information attached to it.

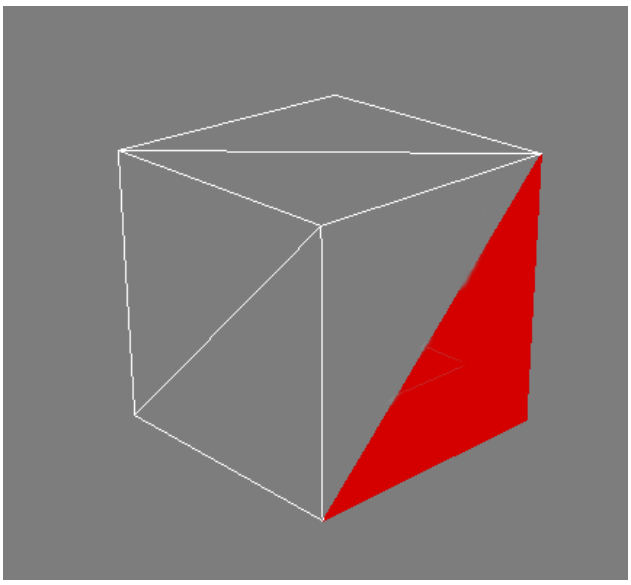


Figure 3.3: Three of the six faces of a cube drawn using triangles.

As many of these vertices contain the same information, we should be able to represent the cube using only 8 vertices, one for each corner. So if we do that and use an IndexBuffer to decide which vertices to use for each triangle, it would contain 36 indices and we could choose to use a very small data type to represent these, for example a byte.

If our vertices only contain a position as three 32bit floating-point numbers, the 36 vertices use 432 bytes (36 vertices * 3 floats at 4 bytes each).

Using the IndexBuffer costs 36 bytes for the indices and 96 bytes for the 8 vertices (totalling 132 bytes). That is a reduction of 300 bytes for a simple cube.

Using an IndexBuffer can cut down on the amount of data we need to send to the geometry stage, allowing us to have more geometry on screen as we are using less bandwidth.

3.2.2 The Geometry Stage

On modern hardware the geometry stage and rasterizer stage are hardware accelerated. Graphics devices are designed to handle the transition from the data we have fed it to displaying it correctly on the screen. It does so very quickly because it has a limited instruction set and is highly specialized and optimized. It also has the benefit of having its task being embarrassingly parallelizable. Each vertex (and later pixel) can be processed independently, which gives the device almost unlimited freedom to balance the workload onto several pipelines running simultaneously.

The first task of the geometry stage is to transform the vertices from *object space* to *screen space* through *world-* and *eye space*. It does so by transforming the position and optionally normals using a concatenation of the three matrices **World**, **View** and **Projection**. The geometry stage can also perform per-vertex lighting calculations, such as Gouraud shading (see [1] for details).

Earlier graphics devices implemented a fixed-function pipeline, which meant the computations done on the device were always the same. The device could be slightly configured for every frame, but once running the computations for every vertex and pixel were the same.

Modern graphics devices have programmable pipelines in addition to the fixed-function approach. With programmable pipelines the developer can make their own shading algorithms and have the graphics device use those during the geometry and rasterizer stage, these programs are popularly known as **Vertex-** and **Pixel Shaders**, also pictured on the overview figure 3.1 on page 15.

The shaders can be written in high-level languages and then compiled for execution on the graphics device, a simple example of a vertex shader function which transforms vertices from object space to screen space written in HLSL (High Level Shader Language) follows:

Listing 3.1: Simple Vertex Shader in HLSL

```
1  // The world, view and projection matrices provided by the Application Stage
2  float4x4 World;
3  float4x4 View;
4  float4x4 Projection;
5
6  // Vertex Shader function, takes a position as input and outputs a transformed position
7  float4 VSTransform(float4 pos : POSITION0) : POSITION0
8  {
9      float4 worldPosition = mul(pos, World);
10     float4 viewPosition = mul(worldPosition, View);
11     float4 screenPosition = mul(viewPosition, Projection);
12
13     return screenPosition;
14 }
```

Once the vertices are transformed, the geometry stage will perform clipping, which is the process of removing the vertices that are not within the *screen space* unit cube, which corresponds to removing all vertices that were not within the view frustum. It will then pass the geometry to the rasterizer.

3.2.3 The Rasterizer Stage

The last stage of the pipeline is to rasterize the geometry; that is convert it to pixels on the screen. The rasterizer stage processes each triangle or primitive pixel by pixel and calculates the correct color of every pixel. It is also at the rasterizer stage textures are applied. By interpolating the texture coordinates on the vertices, the rasterizer can assign each pixel a specific coordinate, which it can use to do a texture lookup on an image. This effectively glues an image onto geometry, an example of which is pictured on figure 3.4.

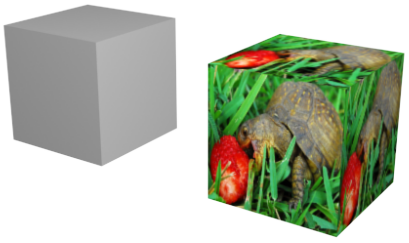


Figure 3.4: The box on the right has a texture applied.

As with the geometry stage, the rasterizer stage is programmable. Pixel shaders can be used on most recent graphics hardware, allowing the developer to apply advanced lighting and effects on a per-pixel basis.

One interesting use of pixel shaders is post-processing effects. By rendering the entire scene onto a texture and then rendering the texture using a quad covering the entire screen, the

developer can use a pixel shader to apply effects to the scene, such as blurring, depth-of-field, distortion filters and much more.

Following is an example of a very simple pixel shader function, which applies a texture to the geometry:

Listing 3.2: Simple Pixel Shader function in HLSL

```
texture Texture;

// The sampler does interpolation between pixels on the texture
// Can be configured to use different algorithms, like nearest neighbor and linear interpolation
sampler texSampler = sampler_state
{
    Texture = (Texture);
};

// The Pixel Shader function, takes a texture coordinate as argument.
// Returns the color from the Texture using texSampler
float4 PixelShaderFunction(float2 TexCoord : TEXCOORD0) : COLOR0
{
    // tex2D is an HLSL function for doing texture lookups.
    return tex2D(texSampler, TexCoord);
}
```

The pixel shader is run on each pixel of each fragment of geometry, which means that if two triangles both overlap the same pixel, the shader function is called twice for that pixel. Usually the color of the last drawn fragment will be the one shown on screen, but the rasterizer on most graphics devices also support using a depth buffer (also called Z-buffer) to determine which fragment is nearest the camera. The rasterizer stage also handles other effects like fog and *alpha blending* (transparency), most of which is hardware accelerated on modern graphics devices[1].

3.2.4 Shading

One of the most crucial parts of creating a believable 3D environment is to apply proper lighting. Modeling real world lighting using brute force is prohibitively expensive, instead approximations are used.

Shading is the process of calculating the color of a pixel, for example using lighting calculations. In games the three most commonly used shadings are **Flat**, **Gouraud** and **Phong**, which correspond to shading per-triangle, per-vertex and per-pixel. Gouraud corresponds to performing the lighting calculations using a vertex shader and Phong corresponds to using a pixel shader. Flat is not available in programmable-pipelines, but a fixed-function pipeline can be set to use flat shading.

The most commonly used *lighting model* to calculate the color in real-time graphics, such as games, is called the Blinn-Phong Model (not to be confused with Phong shading). At each point of shading, three color components are calculated: **Ambient**, **Diffuse** and **Specular**. In the end all three components are summed up to get the final color. See figure 3.5 on the following page for an example.

In this section, the light sources we use are point lights. Point lights have a position from where they emit light in all directions.

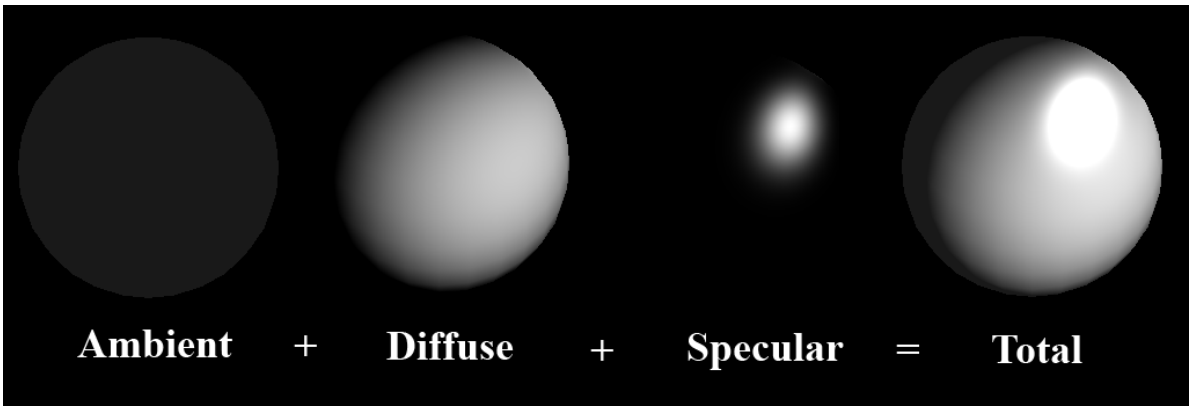


Figure 3.5: The different components of the lighting model. This is rendered using Phong shading (per pixel).

3.2.4.1 Ambient

The ambient component models the ambient light conditions, which is the amount of light that reaches all pixels. This is an approximation for radiosity, which cause light rays to bounce around a room lighting up areas that are otherwise in shadow. The ambient component is not affected by the lights in the scene and is therefore equally applied to all objects.

In practice the ambient component is simply a color multiplied component-wise to the material color. In HLSL:

Listing 3.3: Ambient Lighting

```
// Initialize ambient light to dark gray
// This parameter can be set from the Application Stage
float4 ambient = float4(0.2, 0.2, 0.2, 1);

// matColor is the color of our material at this point.
float4 AmbientComponent(float4 matColor)
{
    // Multiplying in HLSL is component-wise, which means we get:
    // float4(matColor.r * ambient.r, matColor.g * ambient.g, matColor.b * ambient.b, matColor.a * ambient.a);

    return matColor * ambient;
}
```

3.2.4.2 Diffuse

The diffuse component is based on *Lambert's Law*[1], which states: On surfaces that do not shine (ideally diffuse), the amount of light reflected is relative to the cosine between the normal of the surface and the direction from which the light is coming. This corresponds to the dot product between the surface normal n and the light vector l (going from the surface point to the light), if they are normalized.

This means two things: The diffuse light factor is one when the light hits the surface directly, and the factor is zero when the light hits directly from the side (perpendicular to the surface normal). We are not interested in negative factors, because they mean the light is behind the surface.

We calculate the diffuse in HLSL like this:

Listing 3.4: Diffuse Lighting

```
// Bright white light
float4 diffuseLight = float4(1, 1, 1, 1);

// The light's position
float3 lightPosition = float3(10,10,10);

// matColor is the color of our material at this surfacePosition.
float4 DiffuseComponent(float4 matColor, float3 surfacePosition, float3 surfaceNormal)
{
    float3 l = normalize(lightPosition - surfacePosition);
```

```
11
12      // calculate our diffuse factor, ensuring it is 0 or more.
13      float diffuseFactor = max(dot(surfaceNormal, l), 0);
14
15      // we just multiply the factor on our material and light.
16      return diffuseFactor * matColor * diffuseLight;
17  }
```

3.2.4.3 Specular

Specular lighting is meant to simulate the shininess of a surface by creating highlights. It is based on the observation that shiny surfaces reflect light according to the law of reflection ("*The angle of incidence is equal to the angle of reflection*"), in which a ray of light hitting a surface and the light reflected from the surface are angled equally from the surface normal. See figure 3.6. Contrary to diffuse lighting, specular lighting is view-dependent as it changes when the camera changes position. This is why an **eye-vector** is used in the calculations, which is a normalized vector pointing at the camera from the surface point.

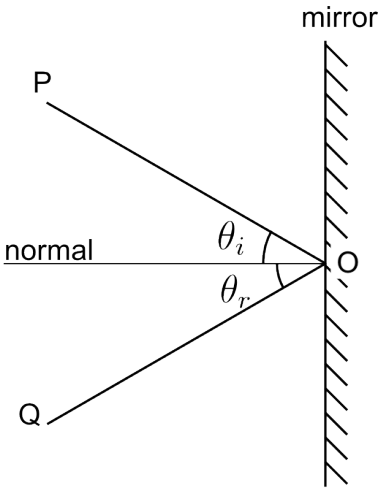


Figure 3.6: Law of Reflection. Image from WikiPedia: `http://en.wikipedia.org/wiki/File:Reflection_angles.svg`

A **shininess** factor is used to attenuate the specular highlights, so it is more prominent where the light is reflected directly towards the viewer.

There are a few ways of creating the highlights. The first uses the angle between the reflection and view vector, as with diffuse, we simply dot the two vectors together and clamp them to the [0-1] range. To achieve the attenuation we take the resulting dot-product to the power of the shininess factor.

Another alternative is to use a *half-vector* and the surface normal instead of using the reflection vector and eye vector. The half-vector is the light vector added to the eye-vector normalized. This approach can be cheaper on architectures that do not have an optimized *reflect* API call. The dot-product of the half-vector and the surface normal is clamped and taken to the power of shininess.

The HLSL source of the techniques are:

Listing 3.5: Specular Lighting

```
1  // Shininess
2  float4 shininess = 8;
3
4  // Specular light color
5  float4 specularColor = float4(1, 1, 1, 1);
6
7  // The light's position
8  float3 lightPosition = float3(10,10,10);
9
10 // The camera's position
11 float3 cameraPosition = float3(-10,10,-10);
```

```
12
13 // Specular using the reflection vector, matSpecColor is the materials specular color
14 float4 ReflectSpecular(float4 matSpecColor, float3 surfacePosition, float3 surfaceNormal)
15 {
16     // Indicent vector (unlike the half-vector approach and diffuse, we need the vector pointing toward the
       surfacepoint)
17     float3 i = normalize(surfacePosition - lightPosition);
18
19     // Eye normal
20     float3 e = normalize(cameraPosition - surfacePosition);
21
22     // Reflection vector reflect is (r = i - 2 * surfaceNormal * dot(i, surfaceNormal))
23     // Taken from MSDN article on reflect: http://msdn.microsoft.com/en-us/library/bb509639(VS.85).aspx
24     float3 r = reflect(i, surfaceNormal);
25
26     float specularFactor = pow(max(0, dot(e, r)), shininess);
27
28     return specularColor * specularFactor * matSpecColor;
29 }
30
31 // specular using half-vector
32 float4 HalfVectorSpecular(float4 matSpecColor, float3 surfacePosition, float3 surfaceNormal)
33 {
34     // Light Vector
35     float3 l = normalize(lightPosition - surfacePosition);
36
37     // Eye normal
38     float3 e = normalize(cameraPosition - surfacePosition);
39
40     // Half Vector
41     float3 h = normalize(l+e);
42
43     float specularFactor = pow(max(0, dot(h, surfaceNormal)), shininess);
44
45     return specularColor * specularFactor * matSpecColor;
46 }
```

3.3 Animation

Most modern games have some sort of animation, implemented either procedurally (in the game code) or authored by animators in a 3D modeling application. This section will describe a few techniques used for animating objects in 3D.

The most basic type of animation used in 3D game engines is **Rigid animation**. Rigid animation is done by manipulating the World transformation of objects over time to achieve motion. It is called rigid because it does not change the fundamental shape of the objects transformed, instead it simply applies linear transformations on the object. Most 3D authoring tools have support for exporting rigid animation.

Animators set up an animation by creating **keyframes** at different points in time for the animated objects and each keyframe is associated a transformation. The application will then do linear interpolation between two keyframes, thereby achieving smooth motion from one pose at a specific time to another pose at a later time. The application can export the keyframes and let the game engine do interpolation or it can simply plot the animation, inserting keyframes for every frame, so the engine will not spend cycles doing interpolation, thus sacrificing space for speed.

3D models are commonly structured hierarchically, such that applying a rotation to a tank will rotate the entire tank, but the turret may be rotatable relative to the tank. This allows for interesting animations and is perfectly acceptable for machines, robots and other inorganic models. However, for organic models such as creatures and humans a more advanced method is required for animations to look compelling.

In animation, **Stitching** is the process of associating vertices with rigidly transformed **Bones**. The bones are placed within an (organic) mesh and each of them are associated with a number of vertices, such that applying a transformation to the bone corresponds with transforming all vertices associated with it. A common application of this technique is to model a humanoid and stitching it to a skeleton of bones to make convincingly animated human characters.

With stitching, a vertex can only be associated with one bone. This is improved with a technique called **Skinning**, where a vertex can be associated with multiple bones with different weight

factors. This is especially important for joints like elbows and knees, where stitching looks very unnatural at large rotations. An example of the two techniques in use is pictured on figure 3.7.

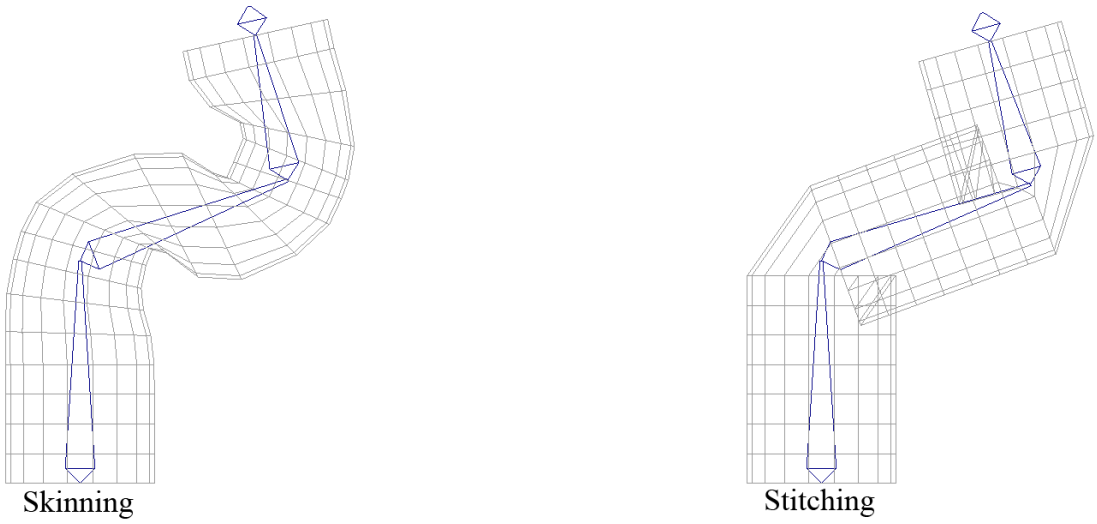


Figure 3.7: Two cylinders with bones, the one on the left uses skinning, the one on the right is stitched.

While **Rigging** a mesh to a set of bones, which is associating vertices to bones for skinning or stitching, the model is in a neutral pose also called the **Bind Pose**. The absolute transformation (in *object space*) of each bone in this pose is used to calculate the position of vertices when the bones are moved, such that the transformation applied to a vertex is the relative transformation between the desired pose and the bind pose. To achieve this we save the **Inverse Bind Pose**, which is the inverted absolute transformation matrix of each bone. When the pose is changed each vertex is transformed first using the Inverse Bind Pose matrix, then the absolute transformation matrix of the bone in the current pose. The two matrices are often concatenated in the Application Stage and sent as one matrix to the Geometry Stage; We call this a **SkinTranform** matrix. An example skinning vertex shader function follows:

Listing 3.6: Skinning Vertex Shader HLSL function

```
1  float4x4 World;
2  float4x4 View;
3  float4x4 Projection;
4
5  // The (inverseBindPose * absoluteBoneTransformation) matrices for every bone.
6  float4x4 Bones[59];
7
8  // We get a position, skinning indices and weights from the Application Stage.
9  struct SkinnedVertexShaderInput
10 {
11     float4 Position : POSITION0;
12     float4 BoneIndices : BLENDINDICES;
13     float4 BoneWeights : BLENDWEIGHT;
14 };
15
16 // Returns the skinned position.
17 float4 SkinnedVS(SkinnedVertexShaderInput input) : POSITION0
18 {
19     // Blend between the weighted bone matrices.
20     float4x4 skinTransform = 0;
21
22     // For skinned animation we add four weighted transformation matrices to get the final transformation.
23     skinTransform += Bones[input.BoneIndices.x] * input.BoneWeights.x;
24     skinTransform += Bones[input.BoneIndices.y] * input.BoneWeights.y;
25     skinTransform += Bones[input.BoneIndices.z] * input.BoneWeights.z;
26     skinTransform += Bones[input.BoneIndices.w] * input.BoneWeights.w;
27
28     // Apply the final transformation to the position
29     float4 weightedposition = mul(input.Position, skinTransform);
```

```
30
31      // Move position into screen space for the rasterizer.
32      float4 worldPosition = mul(weightedposition, World);
33      float4 viewPosition = mul(worldPosition, View);
34      float4 screenPosition = mul(viewPosition, Projection);
35      return screenPosition;
36  }
```

With skinning we reduce the problem of organic animation to the rigid animation of bones, the alternative is to save the position of every vertex at every keyframe and interpolate their positions. This corresponds to having several different versions of the same model that have to be sent through the pipeline, which is often prohibitively expensive in bandwidth.

3.4 Physics

The physics subsystem we need for our game, is a simple 2D collision detection and response system. We also briefly mention the basics of collision detection in 3D.

Collision detection systems in game engines allow checking whether two solid object in the game world collide with each other. Doing precise checks for collision for all objects in the game world would require a lot of computing time, so collision detection systems often use some kind of approximation to speed up the detection of collisions. Often objects in the game world are enclosed in bounding volumes which are then used for collision detection instead of the actual object.

If objects in the game world can be rotated, collision detection using bounding volumes can become imprecise unless the bounding volume is adjusted according to the rotation. Circles and spheres do not have this problem, but if a bounding rectangle is used to represent a rocket in a 2D game and the rocket is rotated, part of the rocket may no longer be inside the bounding rectangle. The problem can be solved in two ways: Either a new bounding volume can be calculated to fit around the rotated object or the bounding volume can be rotated along with the object. Figure 3.8 illustrates this.

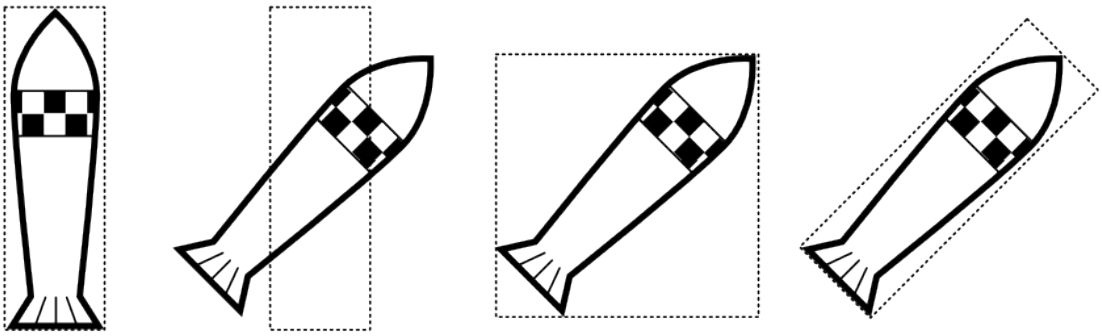


Figure 3.8: Example of orientation of bounding rectangle.

It can be seen from the illustration that calculating a new bounding volume will lead to a less precise bounding volume for the object. However it may sometimes be worthwhile to do this as comparing bounding volumes that are not aligned to the world's axis requires more computing time. A simple workaround for this problem is to represent objects with circles or spheres whenever possible.

Another good optimization is to partition the game world using a spatial data structure. For example, a 2D game world can be partitioned into quadratic and equally sized pieces, called tiles. The idea is to reduce the amount of collisions to detect based on the objects' positions in the world.

Collision Detection in 2D In 2D precise collision detection can be done by doing collision detection per-pixel. Simpler collision detection can be achieved by using bounding circles or rectangles. More complex objects can be represented by polygons.

Collision Detection in 3D In 3D, precise collision detection is extremely complex and therefore used very seldomly. There is a wide range of volumes that can be used to approximate precise collision, where the most often used ones are bounding spheres and boxes. Since 3D collision detection requires more computing power than 2D collision detection 3D games are often designed in such a way that part of the collision detection can be done in 2D.

3.4.1 Collision Response

When a collision is detected in a computer game the game will often have to provide some kind of response. In order to give a response which mimics the real world just detecting whether a collision has occurred is often not enough. The collision detection system in a game which wants to approximate real-life physics will have to provide additional information when a collision occurs, such as where the collision occurred and how the involved objects can be manipulated so that there will no longer be a collision.

3.4.2 Time Stepping

Most physics engines detect collisions each update and then try to solve them according to the physics rules. Some engines use the elapsed time between two updates as a factor in how much to move the objects. One problem with this approach is that fast moving objects might pass through each other between two updates without the system detecting it.

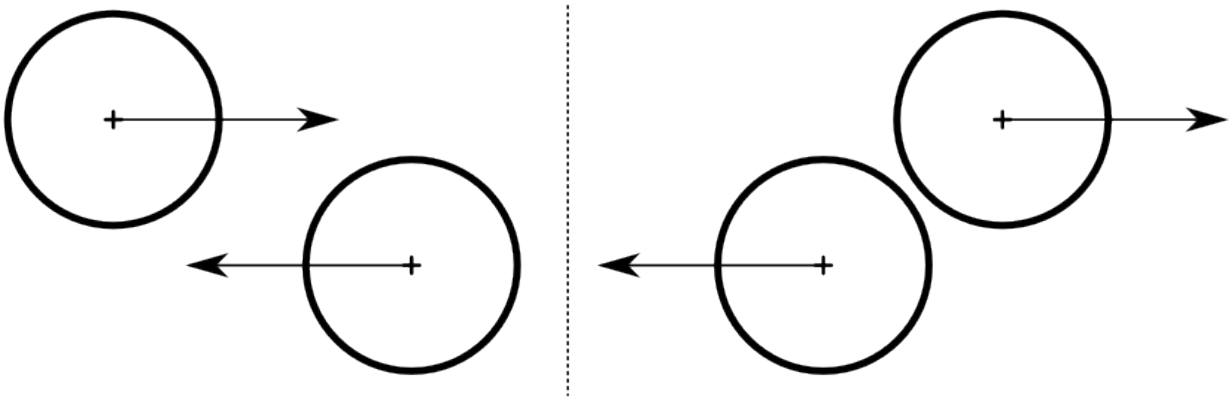


Figure 3.9: Missed collision between two circles.

The left side of figure 3.9 shows two circles moving towards each other at high speed. In the next update (right side of the figure) the two circles has passed through each other. Since the circles do not intersect at the time of either of the two updates the collision detection system will not report a collision between the two circles, even though they have actually passed through each other.

A way to avoid this problem is to do multiple physics simulations in each update in time steps small enough to be sure that no object can pass through another. Another possibility is to not use the elapsed time between updates as a factor and simply move a fixed amount in every update. This causes the physics to be based on how often an update occurs and will slow down the game if the engine cannot keep a stable rate of updates.

Game Engine Selection

In order to select a game engine for implementing the game, there is an overall choice that has to be made. We have to decide on whether to use an existing game engine or to build our own game engine.

There are several arguments for and against both approaches:

Approach A: Use an existing game engine

The primary argument for using an existing engine is that we can start implementing the game right away. By doing this we also get valuable experience on using an existing game engine.

An existing engine could also be accompanied by an SDK enabling content creators to begin work immediately.

Depending on the choice of game engine, the level of required background knowledge of for example 3D graphics can be low, which can increase productivity, since time can be spent on creating the game and not reading up on more theoretical aspects.

Approach B: Build a custom game engine

The primary argument for building our own game engine is that we get valuable in-depth knowledge of how a game engine is constructed. Since we are still going to implement a game, we will still get the experience of using a game engine in practice.

In the process of building a game engine, we will likely encounter a lot of challenging technical issues, which would be already solved in an existing game engine.

Developing a custom game engine for a game can ensure that the engine has exactly the features we need. However, we would need to build these features from scratch, taking up much of the time we could be using on implementing the game.

Furthermore, our engine would not have a preexisting SDK or tools, meaning that these would also have to be developed.

Our choice

In this project we choose to prioritize learning over the extent of the game. In other words we favor the learning process over the extent of the game.

Josh Petrie argues in appendix B on page 97, that in order to create a good game engine it should be based directly on an actual game. Such that the features in the engine are designed to solve specific problems and not hypothetical situations.

Therefore, we are going to implement a working prototype of our game described in the design document from chapter 2 on page 9.

We consider the learning experience from implementing a game engine more important than making a comprehensive, well-polished game. We do not consider any of the engine requirements from section 3.1.4 on page 13 unrealistic for us to implement.

Some of the project group members already have a certain level of experience with game engine development. This experience combined with the technical requirements for the engine, means that we do not see the task of building our own game engine as overwhelming.

In order to get the full experience of using a game engine, we need to build an **SDK**. This is because most existing engines have SDKs of some sort. For our game we want an **editor** to ease creation of levels. This is a big decision because the editor is a separate piece of software with

its own development cycle, and it will take a significant amount of time to build. However, it will provide an interesting technical challenge and a more realistic end product.

The final conclusion of this section is that we are going to implement our own **game engine with SDK** in the form of a **level editor**.

We will decide on platform and technology for the engine and SDK in the following section.

4.1 Choice of Platform and Technology

In this section we will choose platform and technology for implementing our game engine. The requirements for the engine are not unusual or bound to some specific technology, so we have a wide range of options.

All members of the project group own a PC, so for convenience we choose to do the development work on PC and to target the PC platform.

Instead of spending time on less relevant technical details, we would like to work with a level of abstraction which allow us to focus primarily on game engine related challenges. Therefore we need standard libraries with access to graphics device, sound card, file input/output, etc. We are only developing a prototype engine and game, so we prioritize pace of development. This implies the use of a high-level programming language and well-suited standard libraries.

We already have experience with C# and the XNA framework. The XNA framework has the most essential features for use in creating a game engine, and the C# programming language is suitable for creating prototypes, as it is a high-level memory managed language.

A great advantage is that we already know C# quite well, which enables us to be productive early in the development process. We know the XNA framework well enough to ensure that it does not have any technical limitations, which could obstruct the development of our engine.

For our **editor** we can also use C#. Using Windows Forms we will be able to create the Graphical User Interface (GUI) quickly, because we have extensive experience with it.

The next section will elaborate on what the XNA Framework is.

4.2 The XNA Framework

Microsoft's XNA Framework is a set of .NET libraries which allow developers to create games, that will run on Windows, Xbox 360[1] and Zune[2], using C#. The main goal of the XNA Framework is to simplify game development by removing tedious tasks like managing graphics device states, creating a platform window to draw in and managing different input systems. Focusing the developers on making games, not fiddling with native API code. Figure 4.1 on the following page shows the basic structure of the XNA Framework.

[1] Microsoft's gaming console.
[2] Microsoft's portable media device.

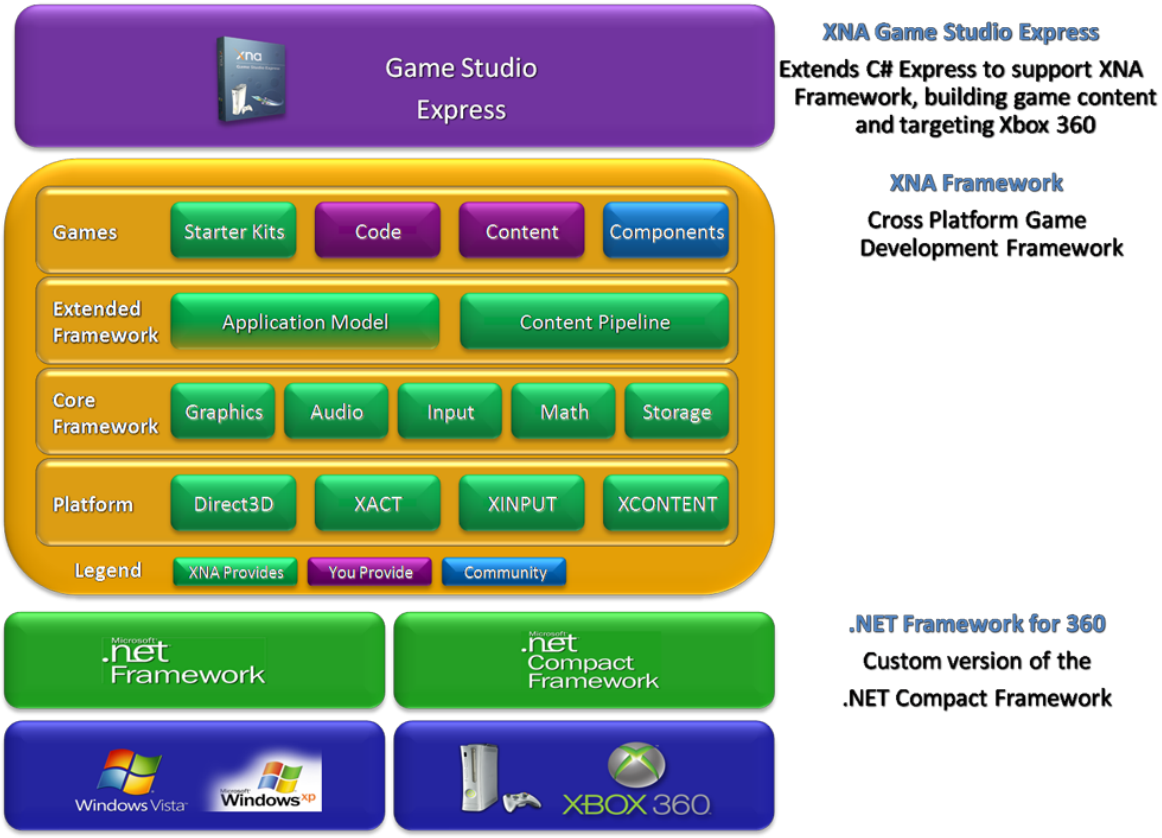


Figure 4.1: XNA Framework Overview. Games are built on top of the XNA Framework, which wraps and abstracts away native platform specific APIs.

XNA Game Studio is the application used to develop XNA games. It is an extension of Microsoft Visual Studio 2008 (for XNA version 3.0) or Visual Studio C# Express 2008. Amongst other features, XNA Game Studio adds the ability to connect to an Xbox 360 or Zune, upload a game and debug it while it runs on the device. Figure 4.2 on the next page shows a running instance of Visual Studio 2008 with XNA Game Studio.

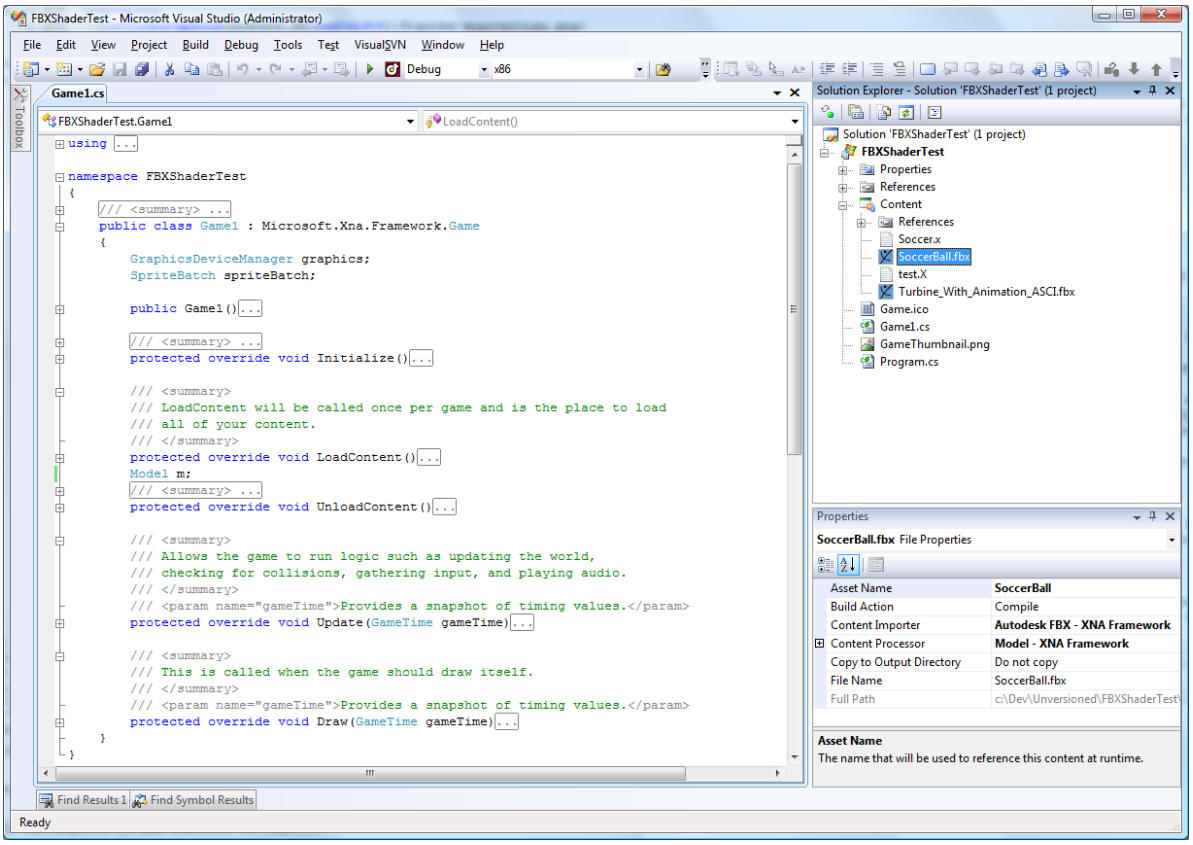


Figure 4.2: Visual Studio 2008 with XNA Game Studio. The selected item in the Solution Explorer to the right is an FBX file included in the Content Project. It will be imported using the Autodesk FBX importer and processed using XNA's Model processor.

4.2.1 Components

The Core Framework consists of a number of .NET namespaces which provide a variety of features. Most prominently it wraps platform specific APIs in managed .NET assemblies for easy access and cross-platform use. It is the standard library on which all XNA games are based, and its main components are:

Graphics wraps Direct3D 9 functionality. It has most of the functionality of Direct3D, but without a fixed-function pipeline. This was left out to encourage use of the programmable-pipeline and because the Xbox 360 does not support it. Furthermore, from Direct3D 10 and forward fixed-function pipeline is no longer supported.

Audio wraps XACT functionality and audio device handling. XACT is a cross-platform audio API for Windows and Xbox 360. Sound authors use the XACT tool to create sound effects, which when loaded into the game can be invoked using simple statements like Play("Explosion").

Input wraps XINPUT which is the cross-platform API which communicates with the Xbox 360 controller and the keyboard on Xbox or Windows. Input also provides easy access and use of the Mouse on Windows. It requires no initialization, to get information about the current state of any input device it is enough to call the appropriate GetState function.

Math provides often used math for games. Such as standard Vector, Matrix, Ray and BoundingBox implementations, as well as helper methods for creating standard Projection and View matrices.

Storage provides the ability to read and write game data, such as save games and highscores, in a uniform way on all platforms. While not very important for Windows development, it is essential for Xbox 360, where it handles all hard disk and memory card access.

The Extended Framework has two parts. One is the Application Model, which abstracts away the platform and handles things like creating a window for the game, providing a game-loop, managing correct timing. The Application Model also has the framework for reusable Game Components. The idea is that the XNA community will develop several components which can be easily shared and reused.

The other part is the Content Pipeline, the purpose of which is to provide a standardized way of handling content, especially 3D content. It consists of a Content Project, to which content is added and configured with an importer and processor. The importer reads and parses content files into standardized content data structures, it can then pass the data to a processor that can do additional processing on the content before serializing it into binary files for the game to read during runtime.

Developers can create their own importers and processors to extend the functionality of the pipeline such as adding support for additional file formats.

A central point of the Content Pipeline is that it does not force people to use specific ways of handling content. Instead of providing a solution which fits all demands, it provides more basic, generic solutions, which can be extended by the users of the framework.

For example, animation support is only partially implemented in XNA. The XNA standard model importer can read the meshes, skeleton and materials, but the standard XNA model processor does not support any output of animation content into the game. This requires the framework users to extend the content processor, and decide on matters like the number of bones supported and support for blending animations. This does require some work, but provides a large portion of flexibility.

CHAPTER

5

The EGGS Engine

This chapter outlines the development of our engine "EGGS" (EGGS is not an acronym) and the main components we developed for it. The first step in developing the engine was to figure out how much was already implemented in the XNA engine and what we could use from it. This is detailed in section 5.1. Then we started implementing the requirements of the engine, each of which has a section in this chapter:

Scene Graph: We need a way to represent objects in our game world. The scene graph will describe the relationship between objects in our world and is a forest (a collection of trees) of **SceneNodes**. The scene graph should also be able to represent information that has no position in the game world. We also need to be able to save and load a scene graph.

Tile Engine: To satisfy the requirement of reusable content we implemented a tile engine. The tile engine will be responsible for updating and rendering our world.

Random Level Generator: We want to randomize the levels in our game. The random level generator creates a new random tile based level.

Lighting and Shading: This section describes the techniques we use for lights and materials in our game world.

Physics: The engine should be able to detect collisions between game objects and against the world. XNA contains no suitable logic for resolving collisions or detecting collision in 2D, so this must be implemented in our engine from scratch.

Animation: XNA has support for importing animation from 3D content files. It does not however have any support for applying and rendering the animation. We need to implement a custom content processor and a way to apply animation in-game.

Screen System: Finally, we need a way of navigating game states. The basic Application Model of XNA has no state management, such as pausing the game and menu navigation. The engine should have a generic system of navigatable screens each with its own draw and update loop.

5.1 What XNA Provides

XNA is a rich framework with many built-in functions we can take advantage of when developing our game engine. This section will describe the features we use extensively.

5.1.1 Application Model

The Application Model as described in section 4.2 on page 27 provides us with a simple structure on which to base our game. When creating an XNA game we are presented with a class that overrides the **Game** class in the XNA Framework.

This class has the following functions:

Initialize For doing any initialization before the graphics device is made ready for loading content.

LoadContent This method is called when the graphics device is ready to load device specific content, for example textures.

UnloadContent When the game ends, this function can be used to unload all the content that is not managed.

Update(GameTime gameTime) is the update loop. It is called once every frame, unless the built-in support for fixed timesteps is used, in which case update may be called several times before Draw. The GameTime object contains information on how much time has passed since last Update was called.

Draw(GameTime gameTime) Is where rendering code goes, the function is called after Update and at the end of the function, the image drawn is shown on screen.

This class also has access to the Content Pipeline using a property called **Content** which is a ContentManager object. The ContentManager has a **Load**$< T >$**(string asset)** function that allows us to load content from the content project. The T is replaced with the type of the content loaded, for example loading a 2D texture is done by:

```
Texture2D someTexture = Content.Load<Texture2D>("someTexture.png");
```

5.1.2 Graphics Device Management

The Game class also has a reference to a **GraphicsDeviceManager**, which manages the device by abstracting away details and providing a simple interface for configuring the graphics device. XNA takes the concepts of managed code from C# into graphics handling, by abstracting away much of the memory management.

In Direct3D, a graphics device could be "lost", which means that the memory on the graphics device was wiped clean, or taken over by another application. For a Direct3D application to continue running when the device is lost, the device had to be reset. Resetting the device means taking control again and setting it back to the settings the application needs. It also means having to reload all device dependent resources, such as vertex buffers and textures, into the memory of the graphics device. The entire process of losing a device, correctly resetting, recreating and loading resources has in XNA been managed, simplifying graphics device management considerably.

5.1.3 Content Import

We take advantage of the Content Pipeline for the content our engine and game will use. XNA already has functions and classes handling the import of 3D models (without animation), textures, bitmap fonts, sounds and more. The content is added to the content project and can be loaded using the ContentManager.

5.1.4 Audio

XNA 3.0 has support for several audio formats and once imported the sounds can be played using a simple function call. We use XNA's built-in functionality for all our sound.

5.1.5 Input

We also take advantage of the input system of XNA. It provides easy access to keyboard, mouse and gamepad inputs, through stateless static classes called **Keyboard**, **Mouse** and **Gamepad**. Each class has a function called **GetState** which we can call to get the current state of all buttons, triggers and directional inputs.

5.1.6 Math for 3D games

XNA has several helper classes and functions for creating 3D games, and we will be using these extensively throughout the engine. This eases the development considerably as implementing this kind of mathematics can be time-consuming, difficult and not as relevant for this semester as knowing how to apply the math correctly.

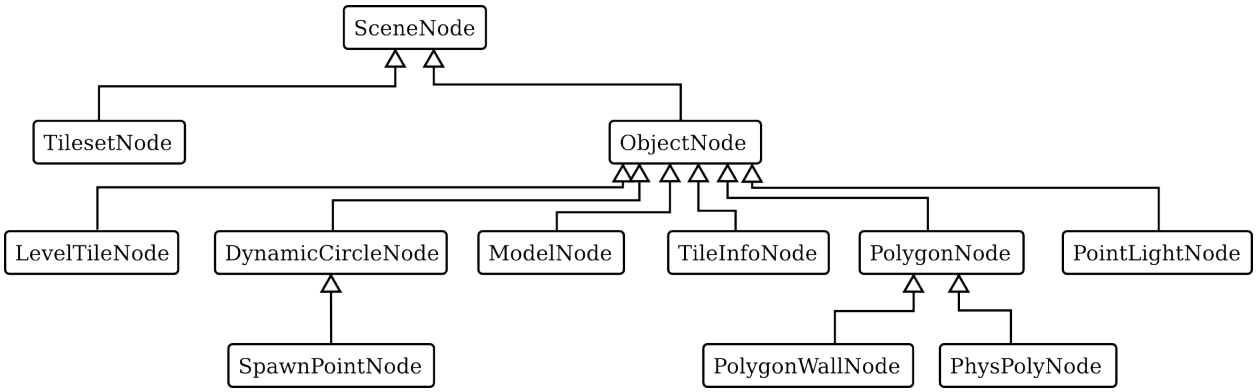


Figure 5.1: The structure of the SceneNodes hierachy.

5.2 Scene Graph

The scene graph is the data structure in which we store our levels. The idea is to have a single format we can use to save and load part of the world. This way a content author can create a scene graph that has all information needed for a player to experience that part of the game, including collision information and graphical content.

The scene graph is a forest (a collection of trees) of **SceneNodes**. All types of nodes extend the **SceneNode** class:

Listing 5.1: A slightly simplified SceneNode Class

```
public class SceneNode
{
    // This SceneNode's parent
    public SceneNode Parent { get; set; }

    // A list of SceneNodes, adding a scene node to this list sets the Parent of the node to this node.
    public NodeList ChildNodes { get; private set; }

    public SceneNode(string name)
    {
        this.Name = name;
        this.ChildNodes = new NodeList(this);
    }

    public virtual string Name { get; set; }

    // The path is constructed using the names of Parent nodes.
    public virtual string Path { get { return Parent != null ? Parent.Path + "." + Name : Name; } }

    // All node types should override this function
    // In this function nodes save themselves in an XmlElement
    public virtual void PopulateXml(XmlElement element, XmlDocument document)
    {
        element.SetAttribute("Name", name);
    }

    // All node types should override this function
    // In this function nodes load themselves from an XmlElement (created using the PopulateXml function)
    public virtual void SetFromXml(XmlElement element)
    {
        if(element.HasAttribute("Name"))
            Name = element.Attributes["Name"].Value;
    }

    // A Tag for game specific purposes
    public object Tag { get; set; }
}
```

A class hierarchy of all types of nodes in the EGGS engine can be seen in appendix 5.1.

5.2.1 Saving a Scene Graph

The graph should be generic enough to be extensible with additional features and nodes in the future, while still being robust enough that changes to the interface of some nodes do not

invalidate all previously saved graphs. Therefore, all nodes have functions for saving themselves into an XML element and loading themselves from an XML element. This simplifies the process of saving an entire forest, as well as providing the kind of backward compatibility we would want for our scene graph. Any elements that are not initialized in **SetFromXml** are simply given a default value, and any properties that have the default value will not be written to the XML in **PopulateXml**. The algorithm for saving the scene graph is as follows:

Listing 5.2: Saving a Scene Graph

```
public static XmlDocument CreateDocument(IEnumerable<SceneNode> node)
{
    // Create an xml document in memory
    XmlDocument document = new XmlDocument();
    document.AppendChild(document.CreateXmlDeclaration("1.0", null, null));
    // add a root element for the graph
    XmlElement root = document.CreateElement("Scene");
    document.AppendChild(root);

    // for every node, recursively add it to the XML file
    foreach (SceneNode n in node)
    {
        root.AppendChild(CreateElement(n, document));
    }

    // Return the populated XML document
    return document;
}

public static XmlElement CreateElement(SceneNode node, XmlDocument document)
{
    // Create an XmlElement for this node
    XmlElement element = document.CreateElement(node.GetType().Name);

    // Use the PopulateXml function to save Type specific information.
    node.PopulateXml(element, document);

    // Recursively create elements for child nodes.
    foreach (SceneNode childNode in node.ChildNodes)
    {
        XmlElement childElement = CreateElement(childNode, document);
        element.AppendChild(childElement);
    }
    return element;
}
```

5.2.2 Loading a Scene Graph

For loading the XML files we use .NET reflection, with which we can instantiate an object of a specific type using only the name of the type. We decided all nodes should be uniquely identifiable in the tree using a name. Thus for our loading algorithm to work, every type of node should have a constructor that takes only the name of the node.

When loading we simply run through all elements in the XML file and attempt to instantiate them using a type that is a SceneNode or a subclass of SceneNode:

Listing 5.3: Loading a Scene Graph from an XmlDocument

```
1  // This is the function we call to load a scene graph
2  public static NodeList LoadScene(XmlDocument loadDoc)
3  {
4      NodeList nodes = new NodeList(null);
5      LoadRecursively(nodes, loadDoc.DocumentElement.ChildNodes);
6      return nodes;
7  }
8
9  public static void LoadRecursively(NodeList targetList, XmlNodeList xmlNodes)
10 {
11     foreach (XmlNode xmlNode in xmlNodes)
12     {
13         XmlElement element = xmlNode as XmlElement;
14
15         // Not an element or doesn't have a Name attribute, then ignore this element
16         if (element == null || !element.HasAttribute("Name"))
17             continue;
18
19         SceneNode node;
20
21         // If we can't instantiate the node, then continue to next xml element
```

```
22          if (!NodeReflector.TryInstantiate(element.Name, element.Attributes["Name"].Value, out node))
23              continue;
24
25          // Set the Properties of this node from using the XML
26          node.SetFromXml(element);
27
28          LoadRecursively(node.ChildNodes, element.ChildNodes);
29
30          // Put it in our list
31          targetList.Add(node);
32      }
33  }
```

The **NodeReflector** which does the reflection can be found in the appendix C.1 on page 99.

5.2.3 Interesting Node Types

This section details a three interesting node types in our engine: The **ObjectNode**, the **PolygonNode** and the **PolygonWallNode**.

Because we want the scene graph to contain information about our game world, we need some nodes to have a position, rotation and scale. The **ObjectNode** has this information and is able to apply its transformation to child ObjectNodes by having an Update function and a World transformation matrix:

Listing 5.4: The ObjectNode Class

```
1  // Inherits from SceneNode and INodeUpdateable which identifies this class as having and Update function
2  public class ObjectNode : SceneNode, INodeUpdateable
3  {
4      public Vector3 Position;
5      public Vector3 Scale = Vector3.One;
6      public Quaternion Rotation = Quaternion.Identity;
7
8      public Matrix World;
9
10     public ObjectNode(string name) : base(name) { }
11
12     public override void PopulateXml(System.Xml.XmlElement element, System.Xml.XmlDocument document)
13     {
14         // Use SceneNode's implementation of PopulateXml to save the objects Name
15         base.PopulateXml(element, document);
16
17         // Do not save the attributes if they have the default value
18         if(Position != Vector3.Zero)
19             XmlHelper.SetAttribute(element, "Position", Position);
20         if(Scale != Vector3.One)
21             XmlHelper.SetAttribute(element, "Scale", Scale);
22         if(Rotation != Quaternion.Identity)
23             XmlHelper.SetAttribute(element, "Rotation", Rotation);
24     }
25
26     public override void SetFromXml(System.Xml.XmlElement element)
27     {
28         base.SetFromXml(element);
29
30         // XmlHelper simply serializes and deserializes a few XNA framework types to and from a string
               representation
31         if (element.HasAttribute("Position"))
32             Position = XmlHelper.ParseVector3(element.Attributes["Position"]);
33         if (element.HasAttribute("Scale"))
34             Scale = XmlHelper.ParseVector3(element.Attributes["Scale"]);
35         if (element.HasAttribute("Rotation"))
36             Rotation = XmlHelper.ParseQuaternion(element.Attributes["Rotation"]);
37     }
38
39     // Declare it virtual to ensure it can be overloaded by other node types.
40     public virtual void Update(GameTime dt, DrawInfo drawInfo)
41     {
42         // Create this node's transformation matrix from its current information
43         World = Matrix.CreateScale(Scale) * Matrix.CreateFromQuaternion(Rotation) * Matrix.CreateTranslation(
               Position);
44
45         // if this ObjectNode's Parent is an ObjectNode, apply the parent's transformation
46         ObjectNode p = Parent as ObjectNode;
47         if (p != null)
48             World *= p.World;
49     }
50 }
```

The **PolygonNode** represents a polygon in 2D, it is the base representation of the static obstacles in our game world. It is a subclass of ObjectNode, so it is possible to position it in the world:

Listing 5.5: The PolygonNode Class

```
public class PolygonNode : ObjectNode
{
    public PolygonNode(string name) : base(name) { }

    private List<Vector2> points = new List<Vector2>();
    public List<Vector2> Points { get { return points; } }

    public override void PopulateXml(System.Xml.XmlElement element, System.Xml.XmlDocument document)
    {
        base.PopulateXml(element, document);

        // Loop through each point and add the point as a child of this XmlElement.
        foreach (Vector2 point in points)
        {
            element.AppendChild(XmlHelper.CreateElement("Point", point, document));
        }
    }

    public override void SetFromXml(System.Xml.XmlElement element)
    {
        base.SetFromXml(element);

        // Run through all the children of this element, if the child is a point,
        // then add it to the points list.
        foreach (System.Xml.XmlElement cNode in element.ChildNodes)
        {
            switch (cNode.Name)
            {
                case "Point":
                    Points.Add(XmlHelper.ParseVector2(cNode));
                    break;
                default:
                    break;
            }
        }
    }
```

The **PolygonWallNode** is a subclass of PolygonNode. It has functions for generating 3D geometry from the 2D points. The idea is that using our editor it should be possible to draw a level using polygons and generate simple geometry from them.

PolygonWallNode creates a wall that runs along the edges of the polygons as well as a cap for the wall. The height of the wall can be configured. The walls are generated using a simple algorithm that first generates four vertices at every point along the polygon, two at floor level and two at the specified height, a quad is created at every edge using a top and bottom of every point. The geometry could be generated using only two vertices at each point, an IndexBuffer and a *TriangleStrip*, but all four vertices of every edge quad must have equal surface normals for the surface to appear flat.

The cap is created using a polygon triangulation algorithm. The algorithm used is called *Ear Clipping*, and is described in Real-Time Rendering[1] along with other triangulation algorithms.

5.3 Tile Engine

A tile engine divides the world into a 2D grid. Each cell of the grid is quadratic, equally sized and is called a tile. Each tile is associated with a scene graph.

In our tile engine there are five different types of tiles, based on the location of exits in the tile. Combined with a rotation the five types describe all configurations of exits possible in a tile. The five types are pictured on figure 5.2 on the facing page.

To make things interesting, we make more than one scene graph for every type of tile and randomize which graph to use for every tile. A collection of these scene graphs used for a level is called a **tileset** and each scene graph is called a **tile asset**.

5.3.1 Data Representation

The tilesets are saved into a single XML file as a scene graph. This allows us to reuse the save- and loading scheme. The tileset scene graph contains a single TilesetNode for every tileset a

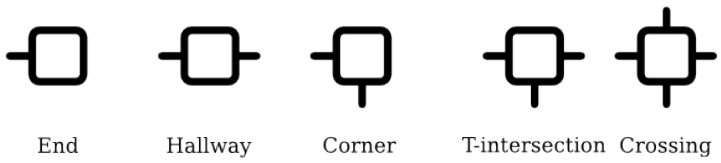


Figure 5.2: The five types of tiles in our tile engine.

game has. A TilesetNode is a subclass of SceneNode and contains three properties:

Height is the default height a tiles in this tileset have. Because the tile-engine is designed for a 2D topography the height is merely a guideline for the content creators responsible for tile scene graphs. (It is used by our editor)

TileSize is the width and height of a single tile. Every tile in the tileset should fit within a square of this size.

ExitSize is the size of the exits in this tileset. Each type of tile has a specific amount of exits that are centered on the edges. This parameter describes how wide the exits should be. An example is pictured on figure 7.8 on page 75.

The XML file is added to the XNA content project and imported using our custom importer called **XmlTilesetImporter**. The importer will read the XML file and look for the tile assets that are associated with each tileset in a subfolder named after the tileset as such:

- Content
  - Tilesets
    - Tilesets.xml
    - <Tileset name>
      - <tile name>.xml

The **XmlTilesetImporter** puts the tile assets and tilesets into a data structure, which will be serialized into an .xnb file for XNA's ContentManager to load during runtime. The import function can be found in listing C.4 on page 104.

For the **XmlTilesetImporter** to correctly identify which tile type a specific tile asset contains, the scene graph must contain a **TileInfoNode**. The TileInfoNode contains various information about each tile asset, such as the tile type, whether or not the tile asset is special (for instance a start or end tile), how to position the camera when the player enters the tile and what texture to use as the floor of the tile.

5.3.2 Loading

Loading is done using the XNA ContentManager, it returns a **TilesetWrapper** object that has the XmlDocuments for all tile assets and tilesets.

Each tile in our tile engine has a reference to a **LevelTileNode**. Each LevelTileNode is then populated using a scene graph from a randomly selected tile asset in the tileset. The **LevelTileNode** is an ObjectNode, which is rotated to fit the rotation of the tile, it also has some optimizations for multi-directional shooters, such as having separate lists for dynamic and static obstacles, as well as baking transformations of static objects in the scene graph.

5.3.3 Updating and Drawing

We need a way to update the tiles and draw them. The **World** matrices of all moving objects must be updated in every frame. Using our tile engine, this is straightforward: We simply iterate through the tiles, calling update on the LevelTileNodes. The LevelTileNode will then update the dynamic objects that are associated with it.

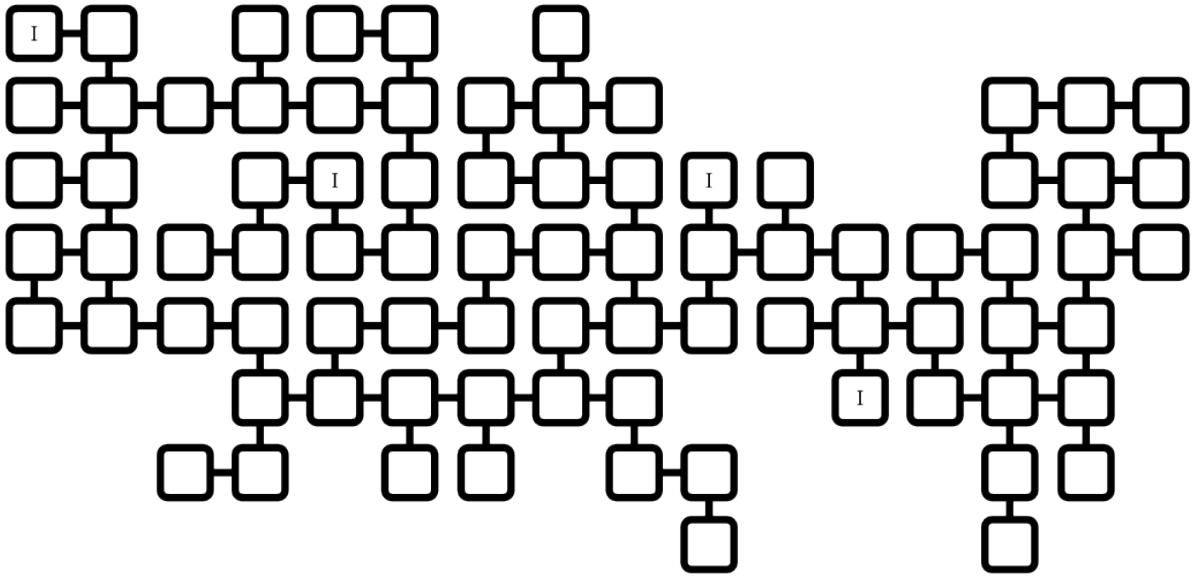


Figure 5.3: The desired output of the level generator - a box is a tile, and a line is an opening between two tiles, an I inside the tile indicates a point of interest.

The same approach is used when drawing, the draw function is called on the LevelTileNodes of interest and they will recursively call the draw function on all all drawable nodes in their scene graph.

5.3.4 Optimization

For our tile engine to be able handle very large levels, attempting to update and render every single tile in every frame would be naive. Instead we need to apply optimization.

Tile engines provide the first and most obvious optimization from its basic design as a spatial data structure, as it divides the world into equally sized tiles. Using only a position we are able to determine which tile in the level we are in. In a multi-directional shooter, we always look down upon the world so there is always a direct relation between the cameras position and which tiles we are able to see. The first optimization is to only update and draw tiles that are around the position of the camera. Using the position of the camera (or player) we choose a tile of interest and only render tiles that are up to a specific *interest* distance from there. Since we represent our level in a two dimensional array we can simply iterate through the tiles that are within the *interest* distance. This way an interest distance of two would draw a $5 * 5$ grid of tiles with the interest tile in the middle.

Associating every tile with an axis-aligned bounding-box, would also allow us to **cull**[1] tiles if their boxes do not intersect with the camera's view projection frustum. Further optimization can be done by associating a bounding volume such as a bounding box to every renderable object in each tile, and cull them using camera frustum culling. We do this for every updated node in every updated tile in every frame.

5.4 Level Generator

A level generator was devised which recursively creates a maze-like graph of connected tiles. A sketch of the desired output of the level generator can be seen in figure 5.3. An important point to note is that every single tile in the level is connected - that is, it is a connected graph. This means that, while playing, no areas of the level are inaccessible to the player.

[1] Culling is the process of discarding unneeded data or material.

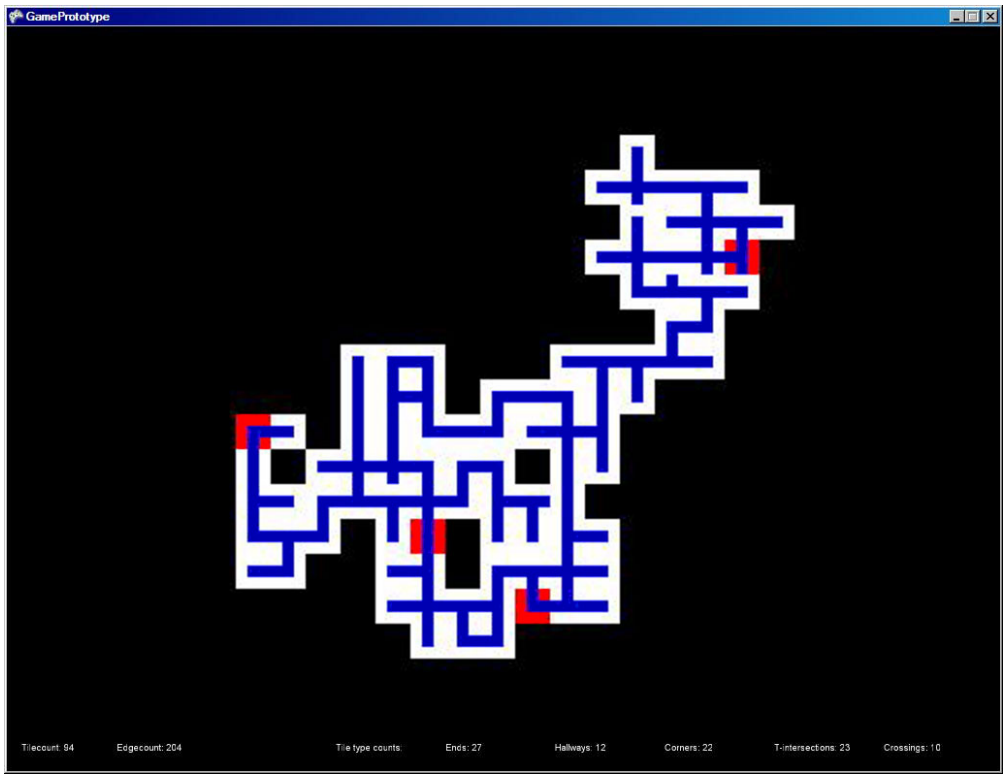


Figure 5.4: The graph of a generated level of size 75 ± 25 and four points of interest.

The screenshot in figure 5.4 shows a visual representation of the output of the level generator.

The level can be seeded with points of interest. These are points inserted randomly with constraints ensuring they will be distributed homogeneously across the maze. The idea is to be able to place special tiles in appropriate positions.

The level generator depends on three vital enumerations to work, as presented in listing 5.6. **Rotation** is used to describe which orientation a tile has. The **TileType** describes the five possible types of tiles shown on figure 5.2 on page 37. **EdgeState** is used internally to describe which of the four edges are passable.

When a **LevelTile** is created, all four edges are marked as **None**, and once the **LevelGenerator** finishes generating a level, all edges are either **Edge** or **Blocked**.

Listing 5.6: The three level generator enumerations.

```
// The numbers are rotation in degrees
public enum Rotation
{
    Up = 0,
    Right = 90,
    Down = 180,
    Left = 270
}

public enum EdgeState
{
    None,
    Edge,
    Blocked
}

public enum TileType
{
    End,
    Hallway,
    Corner,
    Tintersection,
    Crossing
}
```

5.4.1 The LevelTile class

The representation of a tile in the level generator is done through the **LevelTile** class, the signature of which can be seen in listing 5.7.

Listing 5.7: LevelTile class signature.

```
public class LevelTile
{
    public SpecialTileNodeType SpecialType { get; set; }

    public Rotation Rotation { get; private set; }
    public TileType TileType { get; private set; }

    public EdgeState TopEdge { get; set { this.TopEdge = value; UpdateTypeAndRotation(); } }
    public EdgeState RightEdge { get; set { this.RightEdge = value; UpdateTypeAndRotation(); } }
    public EdgeState BottomEdge { get; set { this.BottomEdge = value; UpdateTypeAndRotation(); } }
    public EdgeState LeftEdge { get; set { this.LeftEdge = value; UpdateTypeAndRotation(); } }

    public LevelTile TileAbove { get; set; }
    public LevelTile TileRight { get; set; }
    public LevelTile TileBelow { get; set; }
    public LevelTile TileLeft { get; set; }

    public bool IsPointOfInterest { get; set; }
    public int X { get; private set; }
    public int Y { get; private set; }

    public LevelTile(int x, int y)

    public void UpdateTypeAndRotation()
}
```

When constructing a new tile, it is positioned in the level using a 2D coordinate. The coordinate cannot be changed later.

Changing the state of an **Edge** in the tile, will update the **Rotation** and **TileType** to match the new configuration. The **LevelTile** has a reference to its four neighboring tiles, this is purely for convenience. Finally, **IsPointOfInterest** designates the **LevelTile** as a point of interest.

Level Generation The following listing contains the signature of our level generator:

Listing 5.8: LevelGenerator and Level Information Region Signature

```
public class LevelGenerator
{
    private LevelTile[,] LevelTiles { get; set; }

    // A list containing all tiles in the level.
    public List<LevelTile> CreatedTiles { get; set; }

    // The points of interest in this level.
    public List<LevelTile> PointsOfInterest { get; set; }

    // The constructor which will upon instantiation generate a new level
    public LevelGenerator()

    // A trimmed multi array containing the level
    public LevelTile[,] GetLevelTileArray(int padding)

    // Generates a randomized level
    public void GenerateLevel(int averageTiles, int allowedDeviation)

    // Generates a randomized level, uses the baselevel list as a starting point
    public void GenerateLevel(int averageTiles, int allowedDeviation, List<LevelTile> baseLevel)

    // Creates a LevelTile at the specified coordinates
    private void CreateTile(int xPos, int yPos)

    // Ensures graph sanity.
    private void FindForcedEdges(int x, int y, LevelTile theTile)
}
```

To start generating a new level, we call one of the **GenerateLevel** functions. There are two possibilities available when requesting a new level to be generated: Either we start with an empty level, we use a list of **LevelTiles** and generate a level from these.

The GenerateLevel function will reset the generator and call **CreateTile**. CreateTile is recursive and will end when enough have been generated or it becomes unable to create more tiles. The inner workings of CreateTile is shown in listing C.2 on page 100.

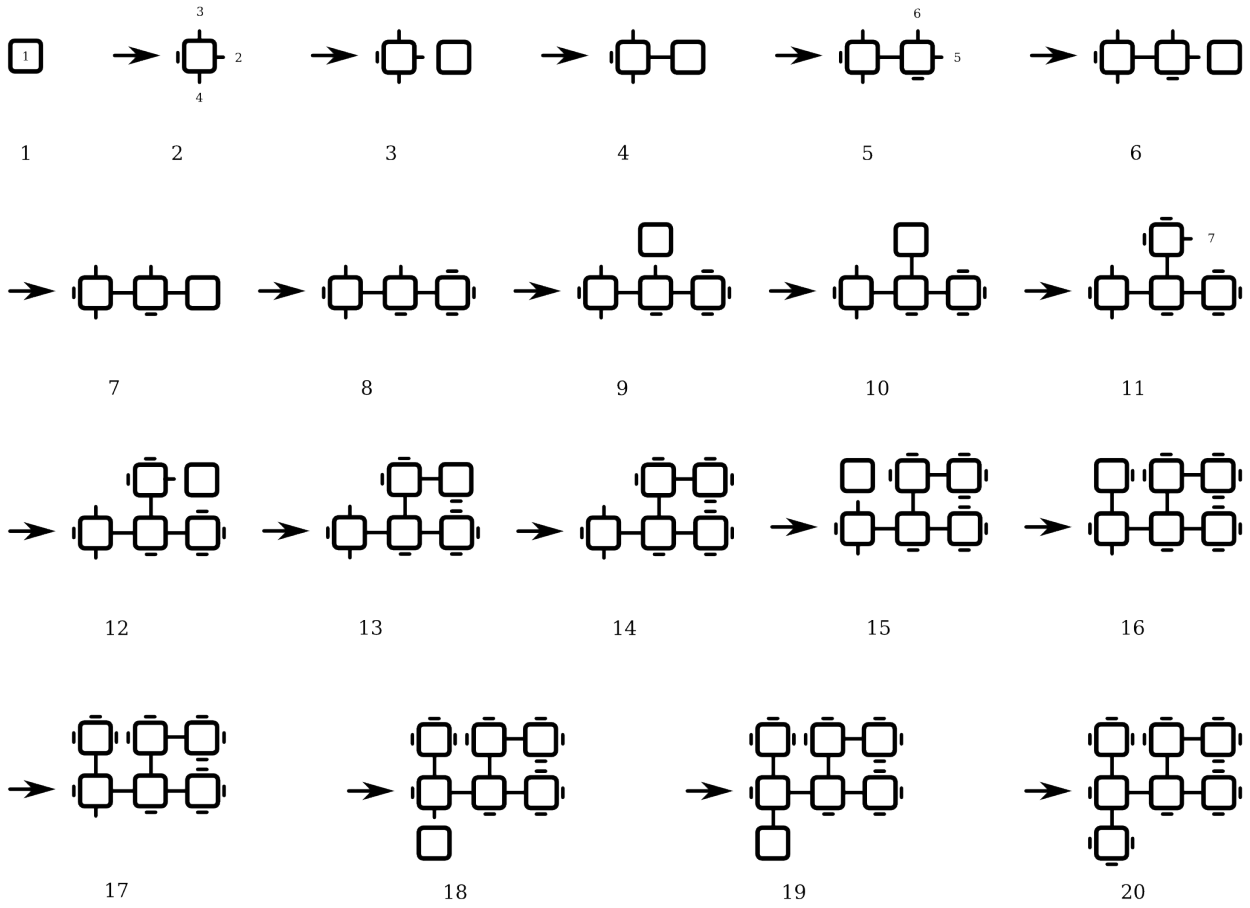


Figure 5.5: A graphical representation of three recursions of the level generator, which in this case results in seven tiles. A line going out of the side of a tile indicates an edge, while a line going perpendicular to the side shows a blocked edge. If two tiles are connected by a line, they are both marked as having an edge going between them.

FindForcedEdges examines the neighboring coordinates around a target tile. If there is a tile at the coordinate the edge going from the target tile to the neighboring tile will be set to the same type as the edge going from the neighboring tile to the target tile.

Sample Run of CreateTile Figure 5.5 shows twenty potential steps of the level generator. The end situation in the figure is either a failed run, caused by randomizer choosing primarily blocked edges, or a completed run with a maximum of 7 tiles, for example using *GenerateLevel(5, 2)*.

GenerateLevel takes the first situation into account when creating levels, by forcing a minimum number of tiles to be created. This is what the **allowedDeviation** argument is used for. The randomization of new edges takes into account that there is a maximum number of tiles available for the level. This is the reason for the randomization of the order in which new edges are assigned, which avoids a level tending to curve in one or the other direction, depending on the order new edges would be assigned in.

Below is a description of each of the twenty steps:

1 CreateTile is called with the coordinates $(0, 0)$ from the GenerateLevel() function. The function creates a new tile in that position.

2 FindForcedEdges does nothing on this first run, as the tile has no neighbors. Step 2 shows the randomization of the tile's four edges - the numbers indicate the order in which the edges are created, they also indicate the order in which new tiles are created (notice blocked edges are not numbered).

3 CreateTile is called recursively, with the coordinate $(x + 1, y)$, meaning the tile immediately to the right of the first one.

4 FindForcedEdges sets the EdgeState of the left edge of tile 2 to Edge.

5 The remaining edges of tile 2 are randomized, with the right side being the first created edge.

6 A new tile along edge 5 is created by calling CreateTile as before with $(x + 1, y)$.

7 FindForcedEdges again sets the left edge of the tile to Edge.

8 The edges on the new tile are randomized, this time resulting in three blocked edges. Ending this recursive branch.

9 Thus returning to tile 2, CreateTile is called on the edge going upwards which is next, and has the coordinate $(x, y + 1)$.

10 FindForcedEdges in this new tile sets the bottom edge to Edge.

11 Randomization of the edges in the tile, resulting in only one new edge being assigned going up.

12 CreateTile is called with $(x + 1, y)$, creating the new tile to the right.

13 FindForcedEdges finds both the edge going towards the tile from the left, but also the blocked edge in the tile underneath, assigning both of those.

14 Randomizing edges assigns blocked edges to all remaining edges, ending the branch.

15 Thus completed all the way along the first of the created edges from tile 1, CreateTile is now called along the next of the edges in the first tile, which is the one above or $(x, y + 1)$.

16 FindForcedEdges finds both the edge going downwards and the blocked edge in the tile to the right and assigns both of those to the correct edges.

17 Randomizing edges finds that all new edges should be blocked and the branch returns to the first tile.

18 CreateTile is called for the tile below the first or $(x, y - 1)$.

19 FindForcedEdges finds only the edge going upwards, and assigns the correct EdgeState.

20 Randomize edges finds that all edges should be blocked in this new tile. All edges are now assigned, and the first CreateTile call in the call stack returns to the GenerateLevel() function.

5.5 Lighting and Shading

For our engine we wanted to explore what we could do with lighting, as it is an effective way to create more compelling visuals without authoring large amount of complex content. We wanted to have several dynamic objects and lights on screen, mainly to investigate the challenges involved in such a rendering system and how to optimize it.

There are three basic ways of having multiple geometries lit by multiple lights:

Single-Pass lighting in which geometry has a single shader, the shader is given a list of lights that affect the object and the shader must light the object correctly in one pass. Because of the limited instruction count of shaders, this technique would only allow us to have a few light sources per object. As Catalina Zima [10] points out, the samples on the XNA Creator's Club [5] can only handle up to 8 lights.

Multi-Pass lighting in which all geometry is drawn once for every light source that affects it. The challenge of multi-pass lighting is to figure out which geometry is affected by which lights and batch them in a way that allows the shader to render multiple lights and geometry in a single pass. The worst case is that the amount of draw calls reaches the number of geometries multiplied with the number of light sources, stressing the geometry stage unnecessarily as the vertex transforms of the objects are done multiple times.

Deferred Shading in which the geometries are drawn once, but instead of applying lighting calculations when drawing the geometry, various properties of the object is written to a number of framebuffers[2]. The properties are variables that are needed to calculate lighting, such as color, normals and position in world space. When all geometries have been drawn, the lights are applied as a 2D post processing effect using the data from the framebuffers.

Of the three techniques **Deferred Shading** is the most interesting. It is very simple to manage as we do not need to worry about which lights hit what geometry, as well as achieving very impressive visuals and requiring us to delve a little deeper into graphics programming than the other lighting approaches. Finally, it requires only one material for every piece of geometry in our world, which means once the basic shader is in place we will not have to make more.

5.5.1 Deferred Shading

Our deferred shading algorithm is based on Shawn Hargreaves presentation at Game Developers Conference 2004[4], and Catalina Zimas article on implementing it in XNA[10].

The main component of a Deferred Shading system is the Geometry Buffer (G-Buffer), which is the collection of framebuffers on which we draw the properties of every piece of geometry. The information we store in the G-Buffer is:

Color The diffuse color of our geometries, this is often just the texture of an object.

Specular Properties For the specular component, we want to save specular power and intensity for every pixel. We use one intensity value for specular instead of a color, for simplicity.

Normal The surface normals of geometries are required for the diffuse and specular components.

Depth By saving the depth (distance from the camera) of each pixel, we can calculate the world position of a pixel using an inverted $View * Projection$ matrix and the screen space coordinates of the pixel. We also need this for diffuse and specular.

When rendering a geometry we write these values to three different framebuffers, called **RenderTargets**. We could render the geometry three times, once for each RenderTarget, but we can also make use of a feature called **Multiple Render Targets (MRTs)**, which allows us to render to all three targets at once. One concern is that only relatively new graphics devices support MRTs (According to Zima[10], this means at least an ATI Radeon 9500 or NVIDIA GeForce from the 6000 series). For this study project this is acceptable, but for production code we would want to implement a fallback to a simpler technique such as *Single-Pass lighting*.

After figuring out what to save, we need to choose how. When using MRTs all our RenderTargets has to have the same bit-depth, which basically narrows the choice of representation down to 32bit, 64bit or 128bit per-pixel per-target. Increasing the precision improves the quality of our lighting, but is also more expensive in both memory and computations. We chose to use 32bit precision, because the techniques we use require no more than that. Were we to implement High Dynamic Range lighting techniques for example, we would have reconsidered switching to 64bit floating point buffers.

For **Color** we use the Color format in XNA which is the standard 8bit for each of red, green, blue and alpha. In the alpha channel we save the specular intensity. For **Normals** we also use the Color format, saving the x, y and z components of the normal in red, green and blue, in the alpha channel we save the specular shininess power. Finally we use the Single format in XNA for depth, which is simply a 32bit precision floating point number.

[2]A framebuffer is an image buffer. It contains an entry for every pixel in an image.

We save values in the color alpha channel because we will not be able to use alpha blending when using a deferred shader. The reason alpha blending is troublesome is that we save the lighting properties at every pixel. When using alpha blended objects we would need information about two different objects at the same pixel: The transparent object and the object behind it. The colors can be blended, but we can only save one normal, and one depth at each pixel.

There are techniques for using alpha blending with deferred shading, but they are beyond the scope of this report. Refer to Shawn Hargreaves presentation for more information[4].

5.5.2 Clearing the G-Buffer

Before we start rendering we want to clear all our RenderTargets by setting their default values. This is done using a simple shader that outputs the same value for every pixel. To render every pixel on screen we use a full-screen quad, just like when applying post processing effects as mentioned in section 3.2.3 on page 18. Although before this is done we need to set the rendertargets as active on the graphics device:

Listing 5.9: Setting the active RenderTargets in XNA

```
// We assume here that the RenderTargets have already been instantiated
private void SetGBuffer(GraphicsDevice graphicsDevice)
{
    graphicsDevice.SetRenderTarget(0, colorRT);     // The Color RenderTarget2D
    graphicsDevice.SetRenderTarget(1, normalRT);    // The Normal RenderTarget2D
    graphicsDevice.SetRenderTarget(2, depthRT);     // The Depth RenderTarget2D
}
```

The shader is a HLSL file we load using XNA's ContentManager. The content of the shader is:

Listing 5.10: ClearGBuffer.fx

```
// Because we are using a full screen quad, we simply pass the position on
// The upper left corner of the full screen quad is (-1, -1, 0) and the lower right is (1,1,0).
// This fits into the screen-space unit cube.
float4 VSPassthrough(float3 Position : POSITION0) : POSITION0
{
    return float4(input.Position,1);
}

// The pixel shader outputs 3 color values, one for each target
struct PSOutput
{
    float4 Color : COLOR0;
    float4 Normal : COLOR1;
    float4 Depth : COLOR2;
};

// The clear function
PSOutput ClearGBuffer()
{
    PixelShaderOutput output;

    // Black for the color buffer, no specular intensity
    output.Color = 0.0f;

    // For normals we are saving a signed value in an unsigned data type
    // [0-0.5[ is the negative range, 0.5 is zero and ]0.5-1] is the positive range.
    // So we set the default value of the normals to 0.5 (or zero :)).
    output.Normal.rgb = 0.5;

    // In alpha we save specular power, this should be zero
    output.Normal.a = 0.0f;

    // Give depth a default value
    output.Depth = 1.0f;

    return output;
}
```

5.5.3 Rendering Geometry to the G-Buffer

Rendering to the G-Buffer is very similar to the usual way of forward rendering (doing the lighting in one pass), except instead of doing the lighting calculations now, we simply save the variables needed to do the lighting. Here is what we do in HLSL:

Listing 5.11: RenderGBuffer.fx

```
float4x4 World;
float4x4 View;
float4x4 Projection;
float specularIntensity = 0;
float specularPower = 2;

// Geometry has one texture, no multitexturing
texture Texture;
sampler diffuseSampler = sampler_state { Texture = (Texture); };

struct PixelShaderOutput { ... } // is the same as in ClearGBuffer.fx

struct VertexShaderInput
{
    float4 Position : POSITION0;
    float3 Normal : NORMAL0;
    float2 TexCoord : TEXCOORD0;
};

struct VertexShaderOutput
{
    float4 Position : POSITION0;
    float2 TexCoord : TEXCOORD0;

    // Using TEXCOORD for normals and depth allows the hardware to interpolate them between vertices
    float3 Normal : TEXCOORD1;
    float2 Depth : TEXCOORD2;
};

// This function is very similar to the standard geometry stage vertex shader (see section 2.2.2)
VertexShaderOutput VertexShaderFunction(VertexShaderInput input)
{
    VertexShaderOutput output;
    float4 worldPosition = mul(input.Position, World);
    float4 viewPosition = mul(worldPosition, View);
    output.Position = mul(viewPosition, Projection);

    // Pass texture coordinates directly to the PS.
    output.TexCoord = input.TexCoord;

    // Transform the normals into world space
    output.Normal = mul(input.Normal, World);

    // We save the z and w components of our position
    // We will be able to get the actual depth value by dividing z by w
    // But first we want the hardware to interpolate the numbers so we can get a value for each pixel
    output.Depth.x = output.Position.z;
    output.Depth.y = output.Position.w;
    return output;
}

PixelShaderOutput PixelShaderFunction(VertexShaderOutput input)
{
    PixelShaderOutput output;

    // Get the diffuse texture color.
    output.Color = tex2D(diffuseSampler, input.TexCoord);

    // Set the specular intensity (the value can be set from application stage)
    output.Color.a = specularIntensity;

    // Save the normal, but first we must renormalize it and move it from the [-1,1] range to [0,1]
    output.Normal.rgb = 0.5f * (normalize(input.Normal) + 1.0f);

    // Set the specular power (the value can be set from application stage)
    output.Normal.a = specularPower;

    // Output the actual depth value of this pixel.
    output.Depth = input.Depth.x / input.Depth.y;

    return output;
}
```

This is also where we would want to add **Skinned Animation**. We could add this as an extra technique and use that technique when rendering skinned geometry. Refer to section 3.3 on page 22 for further explanation, as this method is directly applicable to this shader.

5.5.4 Applying Lighting

Once we have finished drawing everything to our G-Buffer, we are ready to start lighting the scene. This is the most interesting part of the Deferred Shading technique.

We will be using local point lights (see section 3.2.4 on page 19) that have linear attenuation, which means they lose intensity proportionally to the distance to the lit surface.

We apply lighting by first drawing all the lighting calculations to a RenderTarget and then combining the light buffer with our color buffer for the final image. The contributions of each light is added to the final light buffer one at a time, this means we need to set the graphics device to add the colors of overlapping pixels, instead of replacing the current color which is the standard behavior. For more information on *Alpha Blending* and additive rendering refer to Real-Time Rendering[1].

To avoid having to process all pixels on the screen for every light, we use a sphere model to represent a light. The sphere is drawn in world space at the location of the light with a size that corresponds to the reach of the light. This way we will only process the pixels that have a chance of being hit by our lights. This also has the effect that drawing a lot of small lights spends as much time in the rasterizer stage as drawing one big light which covers the same amount of pixels.

Before we draw the lights we need to resolve our G-Buffer, by resolving the buffer we get three textures that have the contents of our framebuffers. We send these textures along with the settings of our light to the point light shader. The following piece of HLSL code is the pixel-shader for drawing a single point light.

Listing 5.12: Pixel-Shader of PointLight.fx

```
// This is an inverted View * Projection matrix
// We use this to go from screen-space back to world-space
float4x4 InvertViewProjection;

float3 lightPosition;   // The position of the pointlight
float radius;   // The reach of the point light
float3 Color = 1.0f; // Color of the light

// Now we need our G-Buffer
texture colorMap;
sampler colorSampler = sampler_state { Texture = (colorMap) ... };

// NOTE: The normalSampler and depthSampler are set to use nearest neighbor
//       It would not make any sense to interpolate and could skew our results.
texture normalMap;
texture depthMap;
sampler normalSampler = sampler_state { Texture = (normalMap); ... };
sampler depthSampler = sampler_state { Texture = (depthMap); ... };

// Is the width of half a pixel as set from the Application Stage
// halfPixel = new Vector2(0.5f/(float)lightRT.Width, 0.5f/(float)lightRT.Height);
float2 halfPixel;

// screenPosition is the interpolated screen-space position of the current pixel
float4 PixelShaderFunction(float4 screenPosition : TEXCOORD0) : COLOR0
{
    // Retrieve the view unit cube x,y coordinates
    screenPosition.xy /= screenPosition.w;

    // We need to use the screen position to do lookup in our textures
    // The screen coordinates are [-1,1] we change this to [0,1]
    float2 texCoord = 0.5f * (float2(screenPosition.x,-screenPosition.y) + 1);

    // Because texels are read from the center and pixels from upper left corner
    // we need to offset by half a pixel. Refer to Real-Time Rendering for details.
    texCoord +=halfPixel;

    // Sample the normal map.
    float4 normalData = tex2D(normalSampler,texCoord);

    // Transform the normals into [-1,1] again
    float3 normal = 2.0f * normalData.xyz - 1.0f;

    // Get specular information from the normal and color buffers
    float specularPower = normalData.a * 255;
    float specularIntensity = tex2D(colorSampler, texCoord).a;

    // Sample Depth Map
    float depthVal = tex2D(depthSampler,texCoord).r;

```

```
51      // Compute world space position by transforming a screen-space point to world space
52      float4 position = float4(screenPosition.xy, depthVal, 1);
53
54      position = mul(position, InvertViewProjection);
55      position /= position.w; // factor w out again
56
57      // Create the light vector, we need for Phong shading
58      float3 l = normalize(lightPosition - position);
59
60      // Compute the linear attenuation.
61      float attenuation = saturate(1.0f - length(l) / radius);
62
63      // Compute the diffuse light
64      float diffuseFactor = max(0,dot(normal, lightVector));
65      float3 diffuseLight = diffuseFactor * Color;
66
67      // Compute specular lighting using the reflect vector method
68      float3 r = normalize(reflect(-lightVector, normal));
69      float3 eyeVector = normalize(cameraPosition - position);
70      float specularLight = specularIntensity * pow(saturate(dot(r, eyeVector)), specularPower);
71
72      // Apply attenuation and return. Notice we do not apply the specular light yet!
73      return attenuation * float4(diffuseLight.rgb, specularLight);
74  }
```

5.5.5 Putting It All Together

After rendering all our lights we are ready to combine our Color buffer with the Light buffer. We simply draw a full screen quad as we have done before and use this shader:

Listing 5.13: Pixel-Shader of CombineFinal.fx

```
1  float4 PixelShaderFunction(float2 texCoord : TEXCOORD0) : COLOR0
2  {
3      // Get the diffuse color from color buffer
4      float3 diffuseColor = tex2D(colorSampler, texCoord).rgb;
5
6      // Get the light from our light buffer
7      float4 light = tex2D(lightSampler, texCoord);
8
9      // Put it together, modulate the diffuse color with the light and add specular light
10     return float4((diffuseColor * light.rgb + light.a),1);
11 }
```

The final result as well as the normal, color and light buffers can be seen in figure 5.6 on the next page.

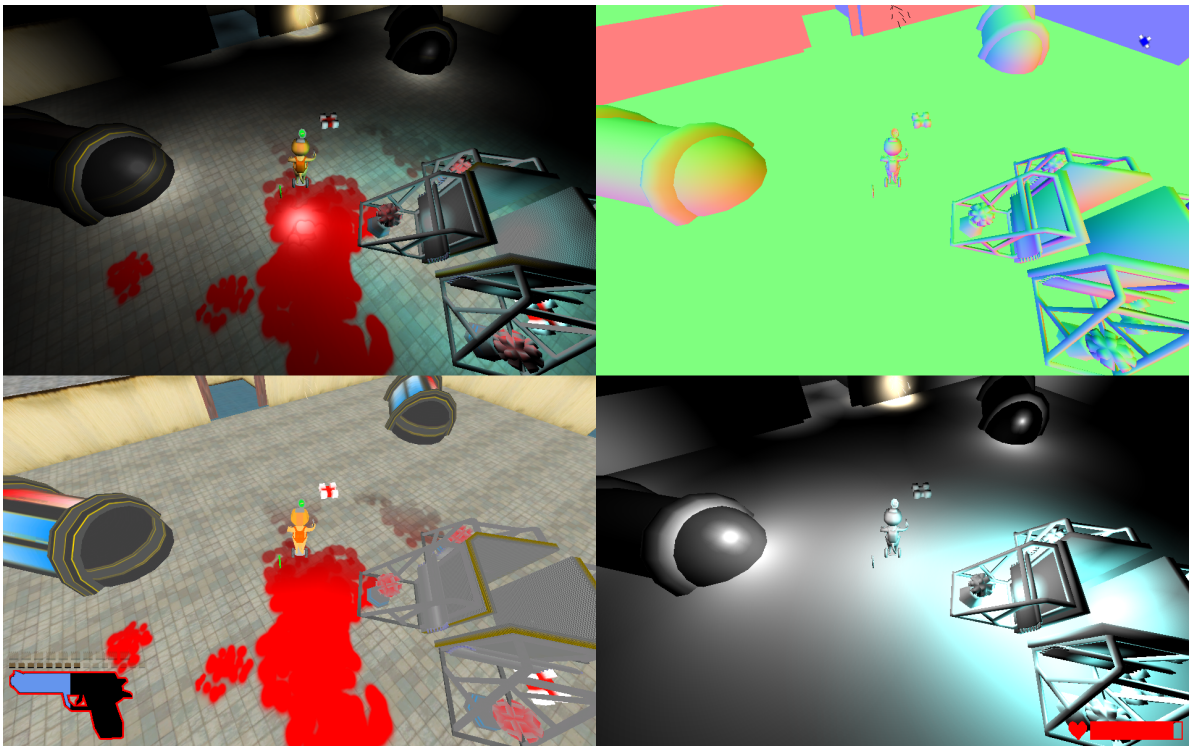


Figure 5.6: Deferred Shading, upper left is the final combination, upper right is the normal buffer, lower left is the color buffer and lower right is the total light contribution from several point lights.

5.6 Physics

The physics system in the EGGS engine is a simple collision detection and response system. The 2D physics system is designed to be used in-game, while the 3D collision detection system is for the editor.

5.6.1 2D collision system

The game engine provides a simple 2D physics system that can detect collision between the following objects:

- Circle/Circle
- Circle/Polygon
- Rays

For Circle/Circle and Circle/Polygon collisions the collision detection system will provide feedback on how to resolve the collision.

5.6.2 Circle / Circle collision

Intersection between two circles can be tested by comparing the distance between the two circle's centers with the sum of their radii. If the sum of the radii is greater than the distance between the two centers the circles must intersect. The collision can be resolved by moving one of the circles until the distance between the two centers is greater than the radii sum. For a more detailed description on how this can be done see David Eberly's essay[3].

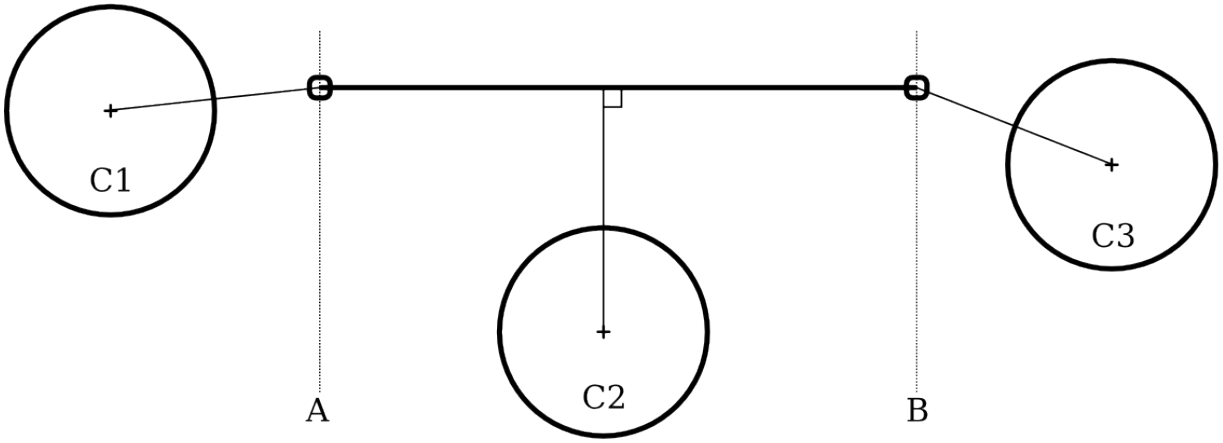


Figure 5.7: Circle/line segment distance.

5.6.3 Circle / Polygon

Intersection between a circle and a polygon is done in the following way: First it is checked whether the center of the circle is inside the polygon. If this is the case there is a collision and the check is done. If the circle's center is outside the polygon the distance from the circle's center to each of the line segments in the polygon is calculated. If any of these distances are found to be between zero and the radius of the circle there will be a collision between the circle and that line segment. The implementation in the game engine speeds things up by making a bounding rectangle around each polygon and checking the circle against this before doing the relatively expensive check against the whole polygon.

The **circle inside polygon** test is done by shooting a ray out from the circle's center and check how many of the polygon's line segments it crosses.

The idea behind this approach is that a ray starting outside the polygon will have to cross the polygon wall an even number of times, one when it enters, one when it exists, and two for each time it exits and enters the polygon again. Since the ray goes on for infinity it is impossible for it to enter the polygon without ever exiting it again and the number of crossings will always be even. However if the ray starts inside the polygon it will not get a crossing from entering the polygon and the number of crossings will therefore always be odd. It can therefore be decided whether the circle's center is inside the polygon by making a ray that starts at the circle's center and count how many of the polygon's line segments it intersects. The ray/line segment tests needed to do this has been sped up by choosing the direction of the ray to be along the positive X-axis, allowing for a lot of early out cases in the ray/line segment test.

Figure 5.7 illustrates the 3 cases that must be considered when calculating the **circle/line segment distance**: When the circle's center is before the start point of the line segment (C1), between the start and end point (C2), or after the end point (C3). To figure out which is the case, a vector from the start point of the line segment to the circle's center is calculated and projected onto the vector from the start to the end point of the line segment. If the length of the projection is less than zero, the circle's center is before the start point. If the length of the projection is greater than the length of the line segment the circle's center is after the end point. Otherwise it will be between the start and end point.

If the center of the circle is before the line segment the distance is calculated by taking the distance from the start point of the line segment to the circle's center and subtracting the radius. Distance to the end point is calculated in a similar manner. If the circle's center is between the start and end point the distance is found by making a vector from the line segment's start point to the center of the circle. This vector is then projected onto the normal for the line segment. The distance is then found by taking the length of this projection and subtracting the circle's radius.

When calculating circle/line segment distance in a polygon/circle intersection test a bit of optimization can be done. If the circle's center is found to be after the end point of the line segment,

the distance test can be skipped, since the end point of the current line segment will also be the start point of the next segment, and the test will therefore be carried out there if necessary.

The engine's collision system resolves polygon/circle collisions by moving the circle until the collision has been resolved. If a collision was found with a line segment start or end point the direction of the resolving movement will be along the vector from the colliding point to the circle's center and if the collision was with the line segment itself the direction will be along the normal to the line segment. How far the circle has to move can be calculated from the distance checks used in the collision detection.

5.6.4 2D Rays

Ray collision testing has been implemented according to the theory found in David Eberly's essay[3]. We can test a ray against a circle or polygon. The system can provide feedback on where the ray collision occurred and the normal to the edge hit at the collision point.

5.6.5 3D collision system

The game engine provides collision detection for the following objects in 3D:

- Frustum/point
- Frustum/Oriented Bounding Box
- Ray/Oriented Bounding Box
- Oriented Bounding Box/Oriented Bounding Box

The intersection tests used in 3D collision detection has been implemented according to the theory found in Essential Mathematics for Games & Interactive Applications[7].

5.6.6 Tile Engine Collision System

The engine can do collision detection and resolving for dynamic and static objects in a tile based 2D game world.

Collision handling is done by finding all interesting dynamic objects and for each of them test and resolve collisions with other nearby dynamic objects. Interesting dynamic objects are defined as any dynamic objects in one of the $5 * 5$ tiles square surrounding the tile coordinates which have cameras in them. Nearby dynamic objects are defined as the dynamic objects in an object's own tile or any neighboring tile which the object's bounding circle intersects with. When a collision is detected it is resolved immediately, moving both of the colliding objects before continuing with collision handling.

When all dynamic versus dynamic collision is done each interesting dynamic object will be checked for collisions with nearby polygons. Again nearby polygons are defined as polygons in the same tile as the object itself and in any neighbor tiles that the object's dynamic circle intersects with. Whenever a collision with a polygon is detected it is resolved immediately, moving the dynamic object before collision handling continues.

The collision system also provides an event, **OnCollision**, that is called each time one object is detected to collide with another. The game logic can hook up to this event to perform special actions on certain collisions. The delegate used for the event is shown in listing 5.14. It should be noted that collisions between two dynamic objects, A and B, will be registered twice with the event, once as a collision between A and B and once as a collision between B and A. Collisions between dynamic objects and polygons will be registered with the dynamic object as the first argument and the polygon as the second argument.

Listing 5.14: OnCollision Event

```
public delegate void OnCollision(ObjectNode a, ObjectNode b);
```

5.7 Animation

As mentioned earlier in the section on XNA 4.2 on page 27, it has only partial support for animations. This section examines how to add this functionality. Furthermore, it describes the process of exporting formats supported by XNA from the 3D modeling packages we have used.

The XNA Content Pipeline already supports import of a number of different file formats, most prominently are the DirectX (.x) and Autodesk FBX (.fbx) formats.

For the models to be imported correctly into XNA we have a few requirements for the formats. First, XNA uses a right-handed Cartesian coordinate system with the Y-axis being up. This causes a few problems because by default, an .x file is left-handed with the Z-axis as up, and while FBX is right-handed, it also has the Z-axis as the up axis by default. So we need the exporters to be able to tweak the output to match the coordinate system of XNA. We would also like the exporters to refer to textures and effects by relative paths, so we can distribute them with ease. The exporters should also be able to handle export of animations and skinning information.

5.7.1 Exporting in Autodesk 3D Studio Max

Developed by Autodesk, 3D Studio Max has great FBX support as it supports the newest version: FBX200900. The latest XNA release 3.0 has improved support for this version, including import of multiple textures and settings, as well as correctly importing the settings of shader effects assigned directly from 3D Studio Max. The dialog for the FBX exporter is pictured in figure 5.8.

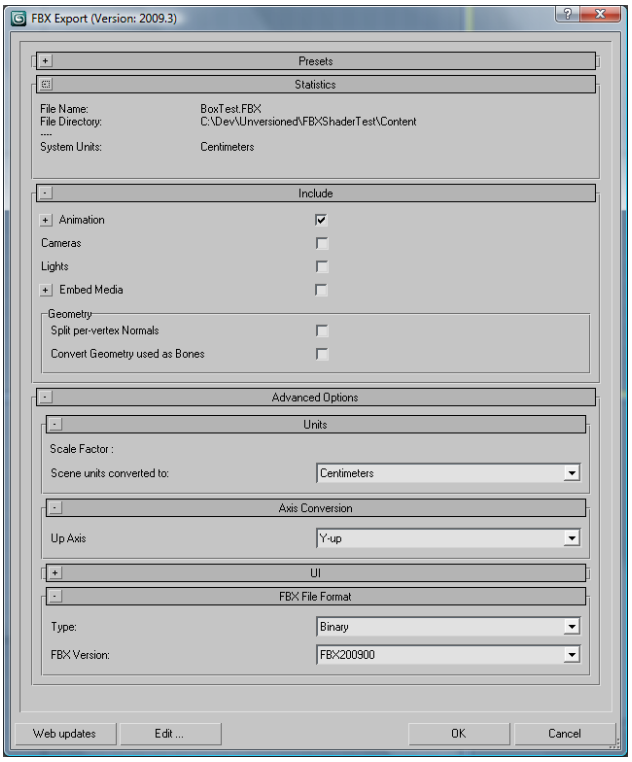


Figure 5.8: 3D Studio Max FBX Exporter Dialog: Example settings for correct export to XNA 3.0.

To ensure the export sizes matches the coordinate system of XNA, we had to set the 3D Studio Max system units to Centimeters. In 3D Studio Max this is done in `Customize→Units Setup→System Unit Setup` where you set "1 Unit = 1,0 Centimeters" using the drop down box. This will probably vary depending on a machine's locale settings. We tested it in XNA by exporting boxes that were $10*10*10$ units, drawing a box of that size in XNA using primitives manually, importing the models and comparing the size.

In the exporter we set the `Up Axis` to "Y-up" and the type to binary. ASCII output is useful for debugging, but there are serious localization issues: On our Danish Windows machines the FBX

exporter uses commas as decimal points, which interferes with the comma separation used to discern numbers, making the export useless.

When exporting animation using FBX, it is only possible to export a single track, but a tool called MotionBuilder[3] can be used to split the single time-line into several animations if needed.

Our engine does not interpolate transforms between keyframes, so we make use of the Bake Animation feature, which creates a keyframe at each frame or otherwise specified step length. By not interpolating at runtime we sacrifice space and fidelity for speed.

3D Studio Max does not have a native .x file exporter. However, there are a number of third party plug-ins available. One is kW X-Port by Jon Watte[8] pictured on figure 5.9, which we used because it is simple and has all the features we need. kW X-port also allows us to have multiple animation tracks without using an additional tool. It does support export of custom shader effects applied in 3D Studio Max like FBX, though it does not support multiple textures on materials. kW X-Port does not use system locale settings for its ASCII export, so floating point number decimals are periods and not commas on our systems. Exporting .x files using kW X-Port is the method we used for most of our models, including our skinned models.

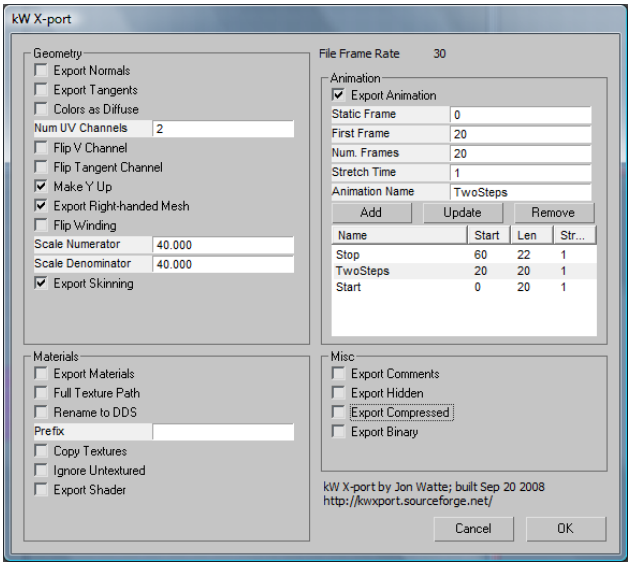


Figure 5.9: kW X-Port: Example settings for correct export to XNA 3.0 with animation and skinning.

5.7.2 Exporting in Blender

Blender supports both .x and FBX with the settings we require. We have only been using blender to export static models without animation, which worked without any incident. Both exporters are built into Blender and are shown on figure 5.10 on the next page. Because Blender is open source it is entirely possible to modify the exporters to any specific needs.

[3] http://usa.autodesk.com/adsk/servlet/index?siteID=123112\&id=6837710

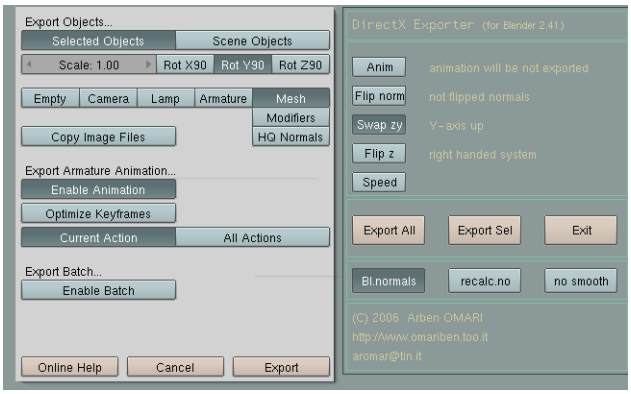


Figure 5.10: Blender Export Dialogs: Example settings for correct export to XNA 3.0.

5.7.3 Exporting in Softimage|XSI

The .x exporter in XSI is very basic. It does not allow you to export the models as right-handed, and it also does not allow you to set the Y-axis as up. This means that to use .x files exported from XSI, we would have to preprocess them before using them in XNA. The exporter is pictured on figure 5.11.

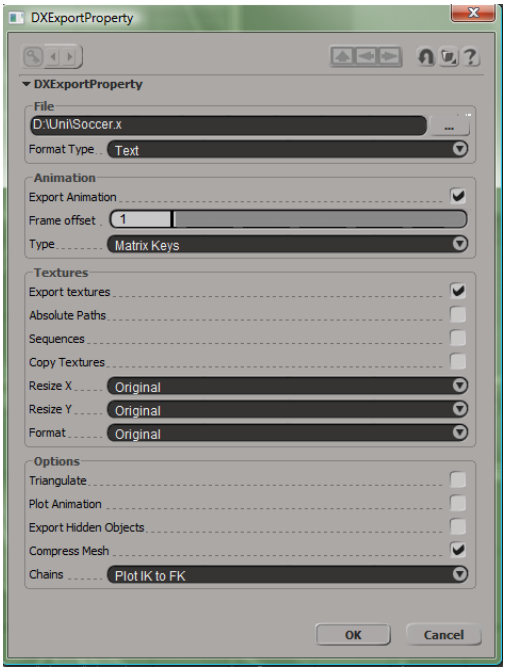


Figure 5.11: XSI .x Export Dialog: Does not allow us to change the handedness to right.

XSI also has an FBX exporter, which is pictured on figure 5.12 on the next page. At the time of writing however, the latest version of FBX supported is FBX200611, which is a mismatch with XNA 3.0. It is possible to import the older FBX versions, but none of the advanced features are supported. The FBX exporter does not offer as many options as the one found in 3D Studio Max, but models exported using it will work correctly with XNA.

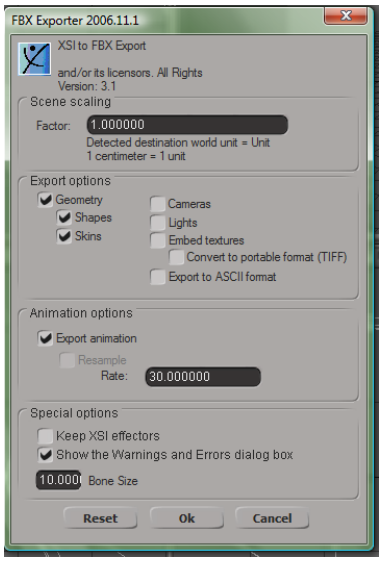


Figure 5.12: XSI FBX Export Dialog: Example settings for correct export to XNA 3.0.

On several occasions we exported models from XSI to OBJ file format and imported those into 3D Studio Max, where we would reapply materials and export to FBX or .x to ensure compatibility with XNA.

5.7.4 Importing Models into XNA

The Content Pipeline of XNA provides us with parsers and simple processors for importing .FBX and .x files. We do not have to worry about the specific syntax of the files, just add the files to the Content Project and select the appropriate importer. The standard processor encapsulates the model data into the **Model** class, which has four properties:

Bones is a collection of **ModelBone** objects, which describe the relationships between the different bones in the model and the ModelMeshes that are related to them. It is what contains the hierarchy of the **Model**, including transformations.

Meshes is a collection of **ModelMesh** objects, which are the renderable parts of the model. Each **ModelMesh** has a number of different **MeshParts** each with their own **Effects** and geometry in the form of a **VertexBuffer** and **IndexBuffer**.

Root is the root **ModelBone**, it is also in the **Bones** collection.

Tag is an object that can be used by custom processors to pass custom data to the game.

This is enough data to render the model and perform hard-coded rigid animation, but it does not provide any rigid or skinned animation data. It is up to us to develop a new processor which extends the basic model processor to include animation data in the **Model** object's **Tag**.

Animation Model Processor The Animation Model processor is a subclass of the standard model processor in XNA. It's main purpose is to take the animation data passed to it from the FBX-Importer or XImporter, format it properly and attach it to the **Tag** property of the **Model** object output. It also attaches correct **Effects** to the different **MeshParts** of the model, here we attach our Deferred Shader to the models.

The main function of an XNA content processor is **Process**. It gets a tree of NodeContent nodes, which is a generic data structure containing model data, as well as a ContentProcessor-Context instance, which is the processor's access to the content pipeline helper functions and environment. The Animation Model processor overloads this function:

Listing 5.15: The Main Function of the Animation Model Processor

```
public override ModelContent Process(NodeContent input, ContentProcessorContext context)
{
    // Find out if any meshes in the three have vertices with skinning weights attached to them
    bool hasSkinning = HasMeshWithWeights(input);

    // Instanciate the container for our animation data
    data = new AnimationData() { Skinned = hasSkinning };

    // The base Processor will return an instance of this class
    // We want to set content.Tag to a dictionary that contains our animation data.
    ModelContent content;

    if (hasSkinning)
    {
        // Get the skinned animation data before letting the base content processor handle the model
        GetSkinnedAnimation(input, null);

        // Let XNA's model processor process the model.
        content = base.Process(input, context);

        // If there were no animations, we log a warning.
        // The skinning data is still useful if we want to do the animations procedurally.
        if (data.Tracks.Count == 0)
        {
            context.Logger.LogWarning("", new ContentIdentity(context.OutputFilename), "Mesh is skinned but has no animation?");
        }
    }
    else
    {
        // For rigid animation, we want to let XNA process the model first.
        content = base.Process(input, context);

        // Then we find the animations and save them in AnimationData.
        FindRigidAnimations(input, content.Bones);
    }

    // We sort all keyframes we have found and use the last keyframe's timestamp as the duration of the entire animation
    foreach (var animation in data.Tracks.Values)
    {
        animation.Keyframes.Sort((x, y) => x.Time.CompareTo(y.Time));
        animation.Duration = animation.Keyframes.Last().Time;
    }

    // Add a dictionary to the tag if it is not already there, then add our animation data to the dictionary
    Dictionary<string, object> tag = content.Tag as Dictionary<string, object>;
    if(tag == null)
        content.Tag = tag = new Dictionary<string, object>();

    tag.Add("AnimationData", data);
    return content;
}
```

The order in which we process animation and the model differs between rigid and skinned animation. For rigid animation we work directly on the ModelBones in the Bones property of the Model without any changes, whereas for the skinned animation we want to get the hierarchy of the bones that are used in skinned animation and put those in a separate list. The bone indices that the XNA model processor applies to the vertices are based on having all skinned bones in one list, separated from the unskinned bones.

The **AnimationData** class contains a map of all animation tracks the file contains, a track contains a list of the keyframes which make up an animation, as well as the duration of the animation. *FindRigidAnimations* simply traverses the model hierarchy and finds all keyframes and puts them into the track map.

In addition to the animation tracks, the AnimationData class also contains all the information needed to perform skinning if the model is skinned. *GetSkinnedAnimation* iterates all skinned bones and saves them and their animations in AnimationData along with the Bind Pose and InverseBindPose of every bone:

Listing 5.16: Saving skinned animation data into AnimationData.

```
public AnimationData GetSkinnedAnimation(NodeContent input, ModelContent model)
{
    // Find the root of the bone skeleton.
    // The skeleton is the same thing as a tree of bones.
    // Only one root is supported in our processor.
    BoneContent skeleton = MeshHelper.FindSkeleton(input);
```

```
7
8       // Bake the transformations of everything but the bones in our skeleton into the model, as the skinned
            bones are the only thing we want to animate
9       // Baking means applying the transformations to the vertices beforehand
10      FlattenTransforms(input, skeleton);
11
12      // Get a flattened list of all bones in our skeleton
13      List<BoneContent> skeletonBones = (List<BoneContent>)MeshHelper.FlattenSkeleton(skeleton);
14
15      // Throw an exception if there were too many bones for the vertex shader function to handle
16      if (skeletonBones.Count > MaxBones)
17      {
18          throw new InvalidContentException(string.Format("Skeleton␣has␣{0}␣bones,␣but␣the␣maximum␣supported␣is␣
                {1}.", skeletonBones.Count, MaxBones));
19      }
20
21      // The bind pose
22      Matrix[] bindPose = new Matrix[skeletonBones.Count];
23
24      // Inverse bind pose
25      Matrix[] inverseBindPose = new Matrix[skeletonBones.Count];
26
27      // A list of indices, its like a map, looking up a bones index in this will return the index of the bones'
            parent
28      int[] skeletonHierarchy = new int[skeletonBones.Count];
29
30      for (int i = 0; i < skeletonBones.Count; i++)
31      {
32          bindPose[i] = skeletonBones[i].Transform;
33          inverseBindPose[i] = Matrix.Invert(skeletonBones[i].AbsoluteTransform);
34          skeletonHierarchy[i] = skeletonBones.IndexOf(skeletonBones[i].Parent as BoneContent);
35
36          // We also save a map of all bones, so we can look them up by name
37          if (!string.IsNullOrEmpty(skeletonBones[i].Name))
38              data.BoneMap.Add(skeletonBones[i].Name, i);
39      }
40
41      // Get all the animations associated with the skeleton and generate animationtracks.
42      foreach (var animation in skeleton.Animations)
43      {
44          AnimationTrack processed = ProcessAnimation(animation.Value);
45          data.Tracks.Add(animation.Key, processed);
46      }
47
48      // Put the data into AnimationData and return it.
49      data.BindPoses = bindPose;
50      data.InverseBindPoses = inverseBindPose;
51      data.SkeletonHierarchy = skeletonHierarchy;
52      return data;
53  }
```

The complete AnimationData object is all we need to add animation to our models at runtime.

5.7.5 Playing Animation At Runtime

To play the animations in game we implemented an **AnimationPlayer** class, it has a reference to the model and therefore also the **AnimationData**. An AnimationPlayer object is needed for every instance of an animated model. This is because the ContentManager ensures that there is always only one instance of a specific Model and AnimationData.

The animation player has three arrays of transformation matrices, the **BoneTransforms**, **WorldTransforms** and **SkinTransforms**, the last is only used for skinned animation. The BoneTransforms matrix array contains the current pose of every bone in the model for this instance, WorldTransforms contain the absolute transformations of every bone (in *object-space*) and SkinTransforms contain the transformations needed by the Skinning vertex shader as described in section 3.3 on page 22.

Once an animation has been started, it progresses by calling an update function on AnimationPlayer with the amount of time to progress, which can be scaled if needed. During the update step the animation player updates the amount of time which has progressed since the animation started. If the animation has ended it can either stop or loop. After time has been progressed, the BoneTransforms array will be updated with values from the active keyframes at the current animation time. Then the WorldTransforms are updated using the BoneTransforms and the hierarchy of the bones from the Model. Finally, if the model is skinned, the SkinTransforms are calculated using the **InverseBindPoses** from AnimationData and the newly calculated WorldTransforms.

5.8 Screen System

XNA provides us with a game-loop, which contains both an **Initialize()**, **LoadContent()**, **UnloadContent()**, **Update()** and a **Draw()** method. This is enough for implementing a game, but for the sake of simplicity of future game code we would like to add more abstraction to the game-loop.

We want a simple way of navigating between states in the game. More specifically we would like a number of game-loops and the possibility to dynamically switch between them. This resulted in the classes **GameScreen** and **ScreenController**, which are described in the next two sections.

5.8.1 GameScreen.cs

The **GameScreen** class can be seen in listing 5.17. It basically corresponds to an XNA game-loop, except that it has a unique ID and a reference to a **ScreenController**, which is responsible for updating and drawing the relevant GameScreens. The ScreenController is described in section 5.8.2.

A thing to notice is that a **GameScreen** has its own **ContentManager**, which makes it simple to avoid having the content of an entire game in the memory at once. A game can be divided into a number of GameScreens, for example by having a **GameScreen** for each level in the game. Level progress could then be handled by loading a new **GameScreen** and unloading the content of the old **GameScreen** from memory.

Listing 5.17: The GameScreen Class

```
public abstract class GameScreen
{
    public string ID { get; private set; } // Used for uniquely identifying a GameScreen
    protected Game Game { get; set; } // A reference to the main XNA game-loop class
    protected ScreenController controller; // Used for navigating between GameScreens
    protected ContentManager content; // Keep references to loaded content

    public GameScreen(string id, ScreenController controller, Game game)
    {
        this.id = id;
        this.controller = controller;
        this.Game = game;
        // Instantiate a ContentManager for use in this GameScreen instance
        this.content = new ContentManager(game.Services);
        this.content.RootDirectory = "Content";
    }

    public abstract void LoadContent(GraphicsDeviceManager graphics);
    public abstract void Initialize(GraphicsDevice graphics);
    public void UnloadContent() { this.content.Unload(); }
    public abstract void Update(GameTime gameTime);
    public abstract void Draw(GameTime gameTime, GraphicsDevice graphics, SpriteBatch spriteBatch, SpriteFont
        spriteFont);
}
```

5.8.2 ScreenController.cs

The **ScreenController** class can be seen in listing 5.18 on the next page. The class is quite simple, so a lot of details are omitted in the code listing.

Performance is essential because the functions of the class are called every frame. All GameScreens are stored in a .NET Dictionary (which is implemented as a hashtable), making it possible to access any **GameScreen** by its ID in $O(1)$ time.

There are three possible states for a **GameScreen**. It can either be both updated and drawn (”Active”), only drawn (”Inactive”) or neither (”Hidden”). To help performance, two lists with references to ”Active” and ”Inactive” GameScreens are used. This means that for example the **Update()** function only needs to iterate through a subset of the screens and update them, instead of checking all screens and only update screens with a specific flag set.

A limitation with the current ScreenSystem design is that it does not make sense to have more than one active and one inactive **GameScreen** at a time. Active screens are drawn in front of inactive screens, but with more screens it is not possible to control the order of rendering. It would be simple to implement it, by for example adding a priority variable to **GameScreen** data and draw screens in order based on that. However, we did not consider it important.

Listing 5.18: The ScreenController Class

```
public class ScreenController
{
    private Dictionary<string, GameScreen> allScreens; // All screens
    private List<GameScreen> activeScreens; // Screens to update and draw
    private List<GameScreen> inactiveScreens; // Screens to draw

    public void AddNewScreen(string id, GameScreen gameScreen)
    {
        allScreens.Add(id, gameScreen);
    }

    public void SetInactiveScreen(string id)
    {
        // Remove screen from active screens, so it doesn't get updated and add it to inactive
        // screens, so its still gets drawn
        GameScreen s = allScreens[id];
        activeScreens.Remove(s);
        inactiveScreens.Add(s);
    }

    public void Draw(GameTime gameTime, GraphicsDevice graphics, SpriteBatch spriteBatch, SpriteFont spriteFont
         )
    {
        // Draw the screens. Active screens must be in front, so they are drawn last
        for (int i = 0; i < inactiveScreens.Count; i++)
        inactiveScreens[i].Draw(gameTime, graphics, spriteBatch, spriteFont);

        for (int i = 0; i < activeScreens.Count; i++)
        activeScreens[i].Draw(gameTime, graphics, spriteBatch, spriteFont);
    }

    // Omitted the following functions: Constructor, LoadScreen(string id), LoadAllScreens(),
    // UnloadScreen(string id), UnloadAllScreens, Update(), SetActiveScreen(string id),
    // SetHiddenScreen(string id) and a few properties.
}
```

CHAPTER

6

The BACON Editor

Our editor for the EGGS engine is named Bad Ass Creator Of Nodes (BACON).

It has two distinct parts: The Windows.Forms based user interface, and the XNA based EditorControl. They are tied together in the code using partly data structure manipulation, and partly a very wide range of C# events, which make it possible to synchronize the status of selected tools in the EditorControl and in the Windows Forms UI. In particular, the ToolHandler, the SelectionHandler and the EditorControl itself contain the most important of these.

As seen in figure 6.1, the editor sits firmly between the Content Importers and a game, and takes care of creating and modifying XML files for scene graphs and tile assets.

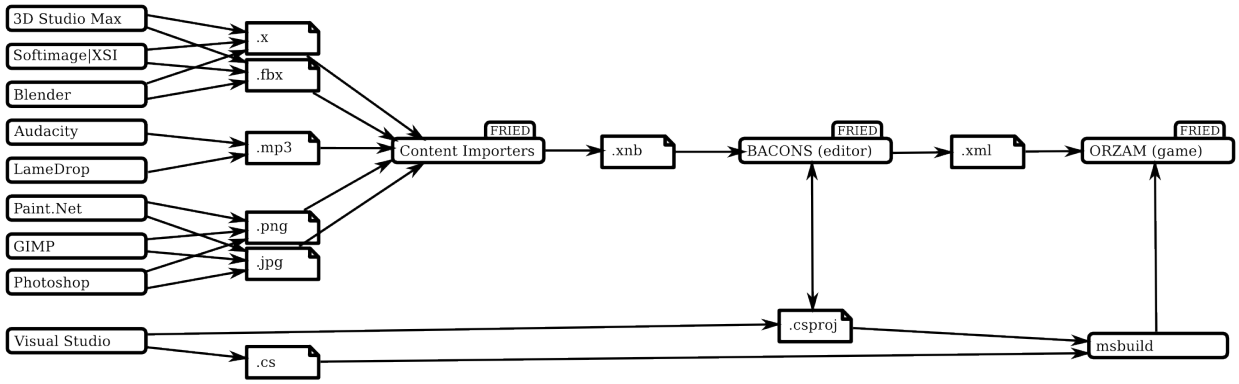


Figure 6.1: The relations between 3rd party programs, the content importers, the editor and the game.

The chapter is separated into three parts: **The GUI** in section 6.1 which describes the graphical user interface of the editor, then **Tools** in section 6.2 on page 61 which describes the six direct manipulation tools available in BACON and finally, **Selected Implementation Details** in section 6.3 on page 63 which contains a walk-through of three of the details editor's implementation.

6.1 The GUI

Before describing some of the code which makes the editor work internally, the elements of the user interface are explained. Figure 6.2 on the following page shows the editor, and the following describes each of the elements in the image.

6.1.1 Toolbar

The toolbar (1) is separated into three sections:

Grid control controls the visibility and fine-grainedness of the two grids in BACON .

Tool selection allows the user to select a tool with the mouse, as well as at a glance see which tool is the currently selected one.

Tool options changes depending on the selected tool. In the screenshot is seen the Select tool's options (consisting simply of a help label), but as can be seen in figure 7.6 on page 73 the

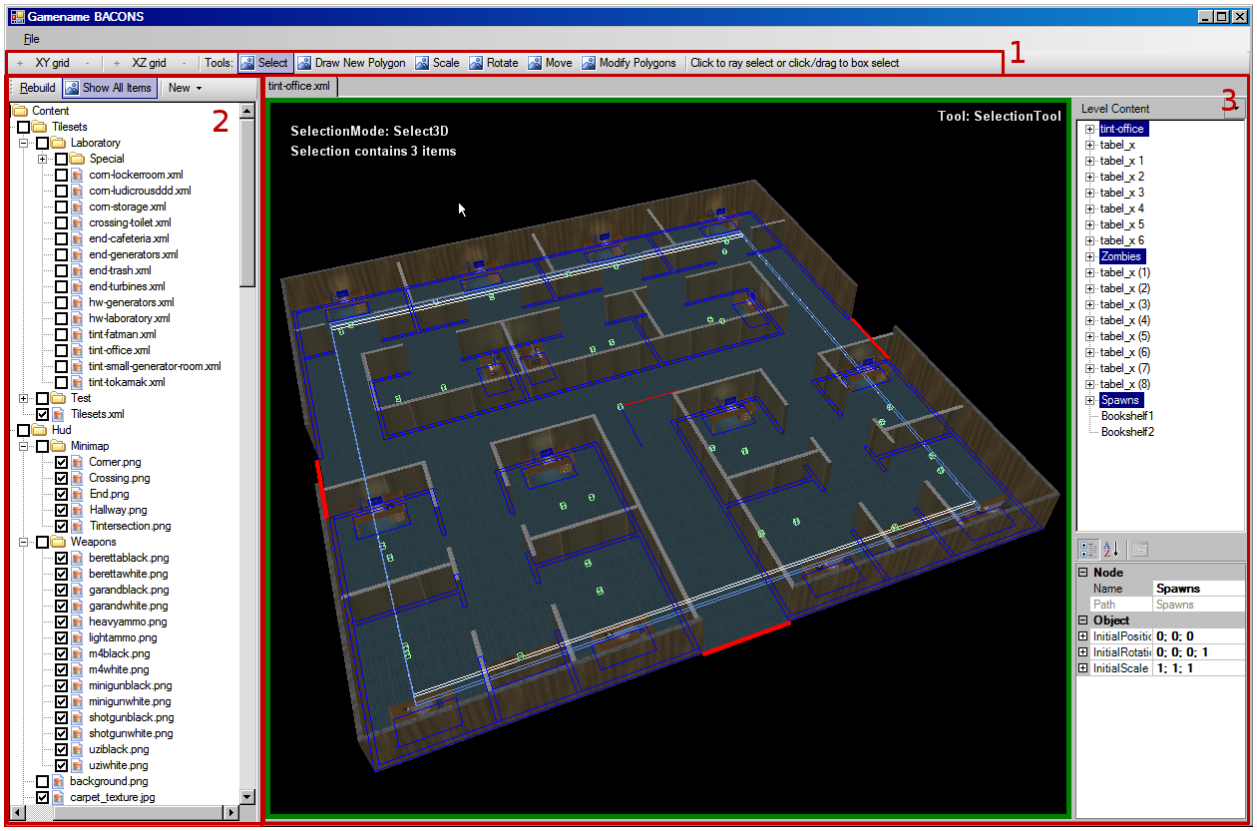


Figure 6.2: The EGGS BACON editor.

Polygon Drawer Tool[1] for example has a dropdown box, which allows the user to select the type of polygon to draw.

6.1.2 Available Content

The treeview on the left side of the window (2) contains all the assets in the Content directory associated with the content project, selected when BACON is started the first time, or through `File→Set Content Project...`.

The `Rebuild` button will call the content builder, which utilizes the Content Importers and Processors discussed previously to build all the content that has been ticked in the tree, and thus make it possible to use that content in the game.

The third button, `New`, shows a menu with two options: New Tile Asset..., and New Scene Graph, which create new tiles asset and scene graph respectively. A further, more intricate description of how this is done can be found in section 7.5 on page 71.

The tree currently operates in two different modes - one showing all supported files found in the Content directory, and another showing only the files currently added to the project. The modes are switched by clicking the `Show All Files` button, and is inspired by the Solution Explorer pane in Visual Studio (see figure 4.2 on page 29, where it is the third button on the Solution Explorer toolbar).

6.1.3 Tabbed Document Section

The third section of the editor window (3) is the document window itself, which uses a tab control to control the currently opened files, either assets or tiles. Closing an open file is done by middle-clicking on the file's tab in the tab bar, or by selecting `File→Close`.

[1] Explained in section 6.2.2 on page 62

6.1.3.1 EditorControl

The EditorControl is a surface upon which XNA draws, and contains all the code which makes the tools function. The thick, green border seen in the screenshot signifies that the mouse focus (and the keyboard focus which follows the mouse) is on the EditorControl - if this turns red, the mouse and keyboard focus is thus elsewhere. What this means is that you can only switch between the different tools using their shortcut keys when this border is green.

6.1.3.2 Level Content Tree

The Level Content Tree display the scene graph of the open file. Any selections made in the tree graph is also made in the EditorControl and vice versa.

6.1.3.3 Property Grid

The property grid shows the properties of the currently selected node.

6.2 Tools

The main functionality of the editor is to create and manipulate objects in a 3D world. For this we need some tools to create objects and apply transformations on these. Implementing such tools is a non-trivial task, as there are many considerations to be made.

The tools should be generic enough to allow the user to create any type of scene graph. The available tools are as follows:

- Selection Tool
- Polygon Drawer Tool
- Scaling Tool
- Rotate Tool
- Moving Tool
- Polygon Modifier Tool

6.2.1 Selection Tool

The selection tools allows the user to select the objects in the 3D world he wishes to modify. It can operate in two modes, one in which it selects positioned objects in the 3D world and one where it selects polygon points. The point selection mode is only active when the selection tool is used as a sub-tool in the polygon modifier tool.

Selection can be done in two ways. The first one is to simply click somewhere on the screen with the mouse. In this case a ray will be shot out from the mouse cursor's position and any objects it intersects will be considered for selection. If no objects have been selected so far the closest of the objects that were hit would be selected. However if one of the objects that were hit was already selected the object behind that one would be selected. This allows the user to select objects behind other objects by clicking multiple times in the same spot.

The other way to select is to hold down the left mouse button and drag the mouse across the screen. This will display a rectangle created with the start and end-position of the drag as top-left and bottom-right corners. A frustum is then created using this rectangle by unprojecting the corners of the 2D rectangle into the 3D game world. The top of the frustum is made by unprojecting the corners a distance of zero and the bottom of the frustum is made by unprojecting to the viewing distance. This way the frustum will start at the camera and cover everything that can be seen within the 2D rectangle. After the frustum is created an intersection test is made between the frustum and all objects in the game world and those that intersect are added to the selection.

All objects in the game world have an oriented bounding box (OBB) associated with them so the intersection test is made as an frustum/OBB intersection test. The frustum is represented by 6 planes and the intersection test is done by doing a plane/OBB distance tests for each plane. Model objects will get their OBB by doing a world transformation of the model's initial axis-aligned bounding box (AABB)[2]. All other objects (such as for example ObjectNodes) get a default $20 * 20 * 20$ axis aligned bounding box.

When an object is selected its bounding box will be drawn to the screen to indicate that it is selected. If more than one object is selected an extra bounding box will be made that surrounds all the selected objects. This bounding box can later be used to quickly check whether the user has clicked on the selection or outside of it.

Due to the hierarchical way objects are stored it is possible to select both a parent object and its children. This might lead to problems when performing operations on the selection: Children who have an ancestor in the selection will get the operation performed twice, once for the ancestor object and once for themselves. To avoid this problem the selection is parsed once the user releases the left mouse button and the necessary child objects are removed from the selection. Following this, the bounding box for all top level objects in the selection is extended so it also surrounds all descendants. The global selection bounding box is also adjusted accordingly. This is to make sure the selection surrounds all objects that will be affected by a transformation. To indicate which descendants have been automatically selected in this way such descendants will have a green bounding box drawn around them.

It is possible to do incremental selection by holding down the control key while making a selection. Incremental selection works as follows: First all objects from the previous selection is added to the selection. Then all objects in the ongoing selection is checked against what is already in the selection. If the newly selected object is already in the selection it will be removed and if it is not it will be added. Incremental selection allows the user to build selections containing objects spread out across the 3D world without having to select objects in between.

Selection of points works in much the same way as selection of 3D objects. The only difference is that point/frustum intersection is used instead of frustum/OBB intersection. To make points easier to hit with ray-selection (single click selection) the points will be considered as small rectangles in the 2D plane. The intersection test is performed between the rectangle and the intersection point of the ray and the 2D plane.

6.2.2 Polygon Drawer Tool

The polygon tool allows the user to draw 2D polygons onto the XZ plane. Polygons are drawn by clicking with the mouse on the points on the screen where the polygon's points are to be placed. When the mouse button is clicked a ray is shot out from the mouse cursor's position and its intersection with the XZ plane is found and used for creating a new point. It is possible to place the camera so that no intersection will be made in which case nothing will happen. While in the process of drawing a polygon the user can right-click with the mouse to end the polygon by adding an edge from the last drawn point to the start point of the polygon. Also if the user tries to create a new point that is close to the starting point a dialog will appear that prompts the user whether he wants to close and finish the current polygon. Created polygons are required to have three or more points. If the user tries to create a polygon with less than three points the attempt will simply be ignored and the tool will reset back to the state before starting the drawing of the polygon.

6.2.3 Scaling Tool

The selection tool allows scaling of selected objects. When the user clicks the left mouse button a ray will be shot out from the cursor's position. If this ray hits the bounding box of the current selection the user can move the mouse up and down to scale the selection. If the selection is not hit by the ray, control is passed on to the selection tool and a new selection is started.

[2]A model is associated with an AABB when it is loaded. The AABB is the smallest AABB that contain all vertices in the model (in *object space*).

If only one object is selected when scaling, that object will simply be scaled around its origin. If multiple objects are selected each object will be scaled around the center of the selection.

By default the scaling tool will scale along all three axes. It is however possible to change which axes will be scaled.

6.2.4 Rotate Tool

The rotate tool works in much the same way as the scaling tool. If the user clicks outside the current selection, control will be transfered to the selection tool and a new selection started. If the user clicks on the selection he can move the mouse left or right to rotate the selection.

If a single object is selected that object will be rotated around its origin. If multiple objects are selected they will be rotated around the center of the selection. By default rotation happens in steps of 45°, but holding down the shift key makes it possible to freely rotate the selection.

By default the tool rotates objects around the Y-axis. The axis to rotate around can be changed by pressing a key.

6.2.5 Move Tool

The move tool works in the same way as the scale and rotate tools when it comes to selecting. If the user clicks outside the selection a new selection is started and if he clicks on the selection he can move the selection by moving the mouse.

By default objects are moved in the XZ-plane. The plane in which to move selection can be changed by pressing a key.

The tool figures out where to move the selection by shooting a ray out from the mouse cursor's position and finding this ray's intersection with the current moving plane. This means that it is good idea for the user always to have the camera positioned so it is facing the plane that is being moved upon.

By default movement is snapped to the grid for the plane that is currently being moved upon. However this behavior can be changed by holding down the shift-button allowing to move the selection freely.

6.2.6 Polygon Modifier Tool

The polygon modifier tool allows the user to modify existing polygons. It operates in two modes: SelectAndMove and AddAndRemove. The user can switch between these two modes by pressing the **ENTER** key.

The SelectAndMove mode works in the same manner as the Moving Tool, except it selects and moves polygon points instead of 3D models.

The AddAndRemove mode allows the user to add and remove points on existing polygons. When this mode is selected a yellow line will be drawn on the screen from the mouse cursor position to the closest point in any of the polygons on the screen. A green line is also drawn from the mouse cursor to the closest of the neighbor points to the closest one. If the user clicks the left mouse button a new point will be added to the polygon from which the lines are drawn, making new edges where the lines were before. If the user clicks the right mouse button the point in the polygon at the end of the yellow line will be removed. It is not possible to reduce the number of points in a polygon to less than three.

Another feature of the AddAndRemove mode is that the normal to the edge between the yellow and the green line is drawn as a white line on the screen. This allows the user to check whether the normals on the polygons are facing the right way. The direction of the normals on the current polygon can be changed by pressing a key.

6.3 Selected Implementation Details

This section describes how certain parts of the editor were implemented - in particular the Property grid, the Build Content Dialog and the way in which the main toolbar and the EditorControl are connected in code.

6.3.1 Property Grid

When the Level Content tree's selection changes the property grid in **EditorPage** is assigned the **SceneNode** represented by the newly selected tree item - a reference to this item is contained in the tree item's **Tag** property. The property grid control then looks at the assigned object, looking for a set of **PropertyGrid** specific attributes, and shows those properties in the grid.

If a value is changed, the **PropertyValueChanged** event is fired. This is caught by **EditorPage**, which takes care of updating the data inside the forest.

If values are changed inside the **EditorControl** itself, using one of the tools there (move, scale...), **EditorControl** calls the **RefreshPropGrid** function to make sure that the property grid data is kept up to date.

6.3.2 Build Content Dialog

When the `Rebuild` button on the content pane's toolbar is clicked, the content project is saved and reloaded, the filelist updated, and finally the **BuildContentDialog** is called. This dialog in turn uses the **XNAProjectWrapper** and **ContentProjectWrapper** classes, which wrap certain functionality of Microsoft's **Build** classes together, to rebuild the content project, showing a progress dialog while doing so, and capturing any errors which might eventually show up. The files involved with this process are Editor/Editor/BuildContentDialog.cs and the four files in Editor/XNAProjects, and though it is not obvious how to implement this type of system, it is still trivial, and as such shall not be described in further depth here.

6.3.3 Toolbar and EditorControl Hookup

To show how the EditorControl is synchronized with the rest of BACON we use the program's toolbar as an example. The same method is applied to the Level Content tree and the Property Grid.

This is important because the hookup does not work as a toolbar normally would. That is, the toolbar control tells the application which tool must now be used, and then expects that to happen, thus setting itself to a state according to this. In stead, the toolbar in BACON sends a request to the EditorControl in the current tab that a certain change is requested, and then does nothing further. If the EditorControl at any time changes its status, it fires an event, which is subscribed to by the toolbar, and acted on accordingly. As an example of this, the process of switching from the Select tool to the Move tool is described next. The code described below is found in EditorControl.cs file and Form1.cs.

The first thing that must happen to get this to work, is setting up the framework for it. What this means is that in the **EditorPage**, an event named **ToolChange** is created.

Form1 subscribes to the **ToolChanged** event in the **hookUpToolbar** function, which is called whenever an **EditorPage** (the control which contains the **EditorControl**) is created, meaning when a scene is created, opened or switched to.

This makes it possible for the **EditorPage** to inform **Form1** of any internal changes to the currently active tool, which is done either on **SetTool** or in the **Update** function, where **EditorPage** discovers which tool is the active one.

The final bit of code which makes the toolbar work is the **toolStrip1_ItemClicked** function, which handles all clicks on the toolbar. All toolbar items have a string assigned to their **Tag** parameter, and if that is the same as the name of one of the tools, the **EditorControl**'s **SetTool** function is called with that name, which sets the entirety of the event chain in motion, eventually causing the toolbar to be updated with this new selected item, and the active subtoolstrip to be assigned through the **editor_ToolChanged** function in **Form1**.

CHAPTER

7

The ORZAM Game

This chapter describes the implementation of our game: Oblivious Rampaging Zombies And Monkeys (ORZAM). It contains the following sections:

Overview: Provides a short overview of the game, containing screenshots and a sales pitch.

Game Architecture: Describes the overall architecture of the game.

Tileset Design: Describes how the different tiles of the game were made; from sketches on a piece of paper, through the use of the BACON editor, to finally being implementable in the game.

Asset Work: Describes work put into the assets and their influence on the game.

Populating the Game World: Explains how we spawn enemies and items in the world.

Movement and Physics: Details how enemies and players move around in the game world.

Special Effects: A description of the different kinds of effects in the game.

Game State Implementation: Describe the different game states and screens used in the game.

Audio: A short description of the game specific audio implementation.

7.1 ORZAM Overview

This section is done in the style marketing departments of game companies often use to describe their games.

The main features of ORZAM can be described as follows:

> *ORZAM is a compelling visual and auditive gaming experience with three action-packed levels in an exciting environment. Playing a level is always an unique experience. Fight against believable computer-controlled resistance using a variety of weapons, including Uzis and Miniguns. Play alone or with up to three friends simultaneously using splitscreen-technology using both keyboard and gamepads. But remember a high degree of player co-operation is needed to defeat innovative game mechanics like yellow morphing zombies flying out of cannons.*

Figure 7.1 on the next page shows action taken directly from in-game. The monkey is fighting for his life against a much stronger enemy, a giant morphed undead brown zombie with an axe in the back of its head. The floor is already splattered with monkey and zombie blood.

Figure 7.2 on the following page shows the truly amazing graphics in the game. Genuine 3D real-time graphics running at high resolution at a stable fixed frame-rate. Also notice the innovative-looking 3D models and their persuasive real-time shaded dynamic static light setting.

Finally figure 7.3 on page 67 shows the flexible multiplayer functionality. Up to 4 players can easily join, using multiple input devices and the opportunity to look at each others screen makes it an unforgettable co-operative gaming experience.

In the next section, we will describe how we made all this possible.

Figure 7.1: Screenshot from ORZAM showing the monkey in heavy fighting with a giant brown zombie, which has just been shot out of a cannon.

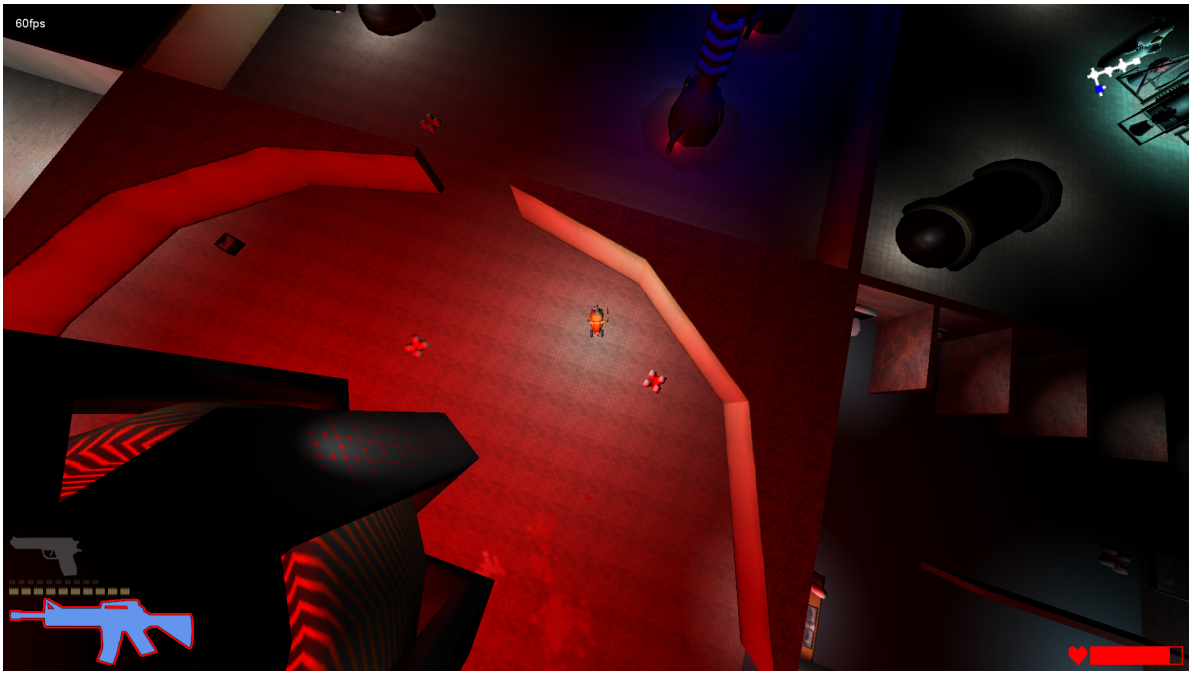


Figure 7.2: Screenshot from ORZAM showing one of the game areas with overview camera control. Notice the lighting around the objects.

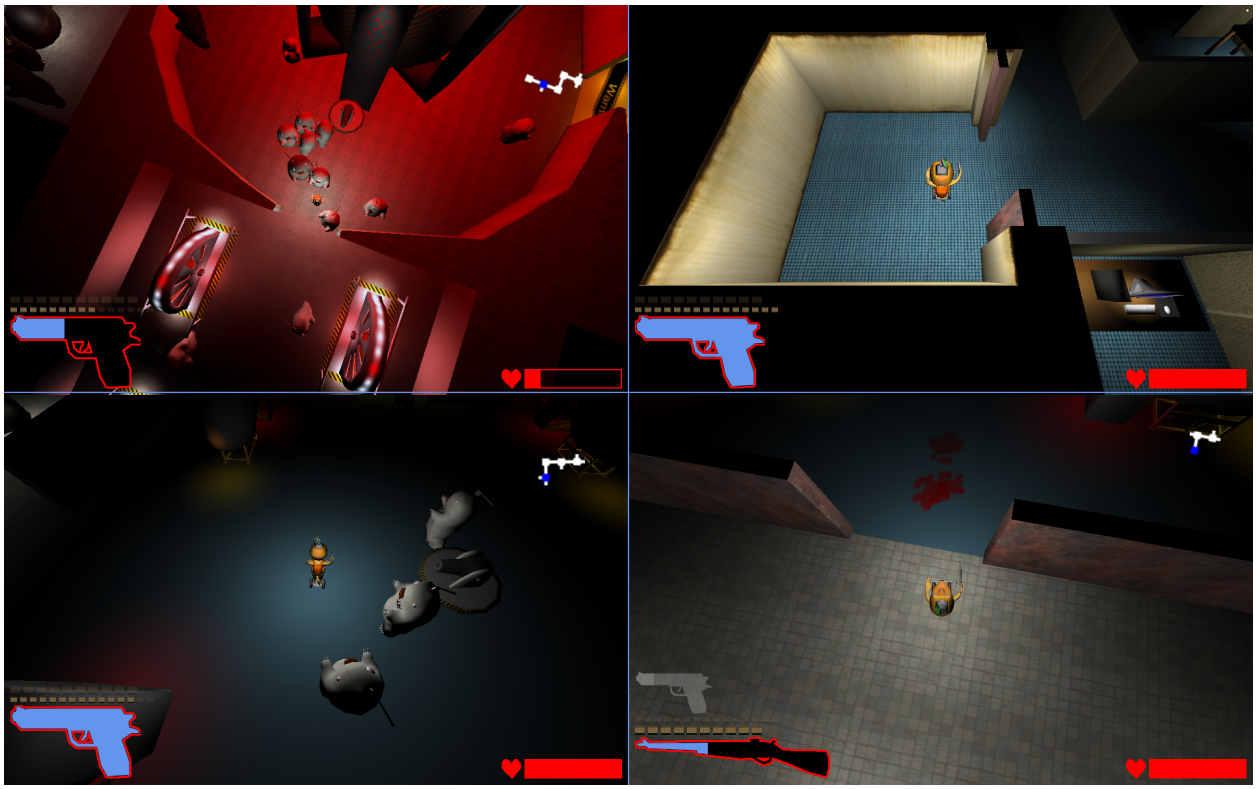


Figure 7.3: Four player split screen action.

7.2 Game Architecture

This section describes the overall architecture of the game logic in the ORZAM game. The game logic is implemented in C# on top of the libraries from the EGGS engine, using an object-oriented approach.

Our main entry point in the application is a class called **Game1**, which contains the game-loop given by XNA Framework. Game1 is merely a container for the Screen System (see section 5.8 on page 57), which extends the game-loop with support for additional game states. The Screen System is used for displaying various graphical user interface screens and navigation. One of these screens run the GameWrapper class, which is the main glue of the game.

The **GameWrapper** handles essentially everything in the game, so instead of going into details, we will provide the pseudo-code seen in listing 7.1.

Listing 7.1: High-level pseudo-code of Update() and Draw() in the GameWrapper class

```
Class GameWrapper
{
    Update()
    {
        UpdateAmbientSounds() // E.g. play bear sounds
        GetInputFromPlayers() // Gather input and move players

        For each player
        {
            CheckIfInGoalTile() // Did anyone find the goal tile?
            CheckIfDead() // If one dies, everyone loses
            UpdateCamera() // Update chase camera according to player position
        }

        UpdateTiles() // Update tiles including things in them
        UpdateEffects() // E.g. advance projectile traces
        UpdateEnemies() // E.g. check if player is nearby and move enemies around
        UpdateCollision() // Find and resolve collisions
    }

    Draw()
    {
```

```
        CheckIfLostOrWonScreenShouldBeDrawn() // Show win/lose screen?

        For each player
        {
            BeginDeferredRendering()
            DrawTiles() // Draw tiles, including static models, enemies etc, in the tiles
            DrawParticleSystems() // Draw projectile traces, wall sparks, etc
            DrawBlood() // Draw the blood splatters
            EndDeferredRendering()

            DrawPlayerHUD() // Draw weapon information and the mini map

            DrawResultingScreenToSplitScreen()
        }

        // Render all split screens to the screen.
        SplitScreensToScreen()
    }
}
```

The **Player** class contains data and functions for the player. In-game a player is visually represented by a model of a monkey on a segway. The Player class extends DynamicCircleNode, which represents dynamic objects in the game.

The Player class functions are mostly centered on weapon control and picking up items. The pickup system is described in section 7.3.

The **Weapon** class handles firing, reloading, sounds, etc. for a weapon. Weapons are created from a static class called **WeaponFactory**, using a factory design pattern. The point of using this pattern is to store all informations the same place, to make it easy to change the properties, for example the damage of a weapon.

7.3 Populating the game world

All non-static objects in the game are placed in the game world at spawn points. Spawn points are placed in each tile as **SpawnPointNode**s using the BACON editor. **SpawnPointNode**s can be one of the following different types:

- Player
- Health
- Ammunition
- BigAmmo
- Weapon
- BigWeapon
- Zombie
- BigZombie
- ZombieLauncher
- Random

7.3.1 Position Type Spawn Points

Position type spawn points are spawn points that simply provides a location where a certain type of object can spawn. Player, Zombie, BigZombie and ZombieLauncher are all Position Type Spawn Points.

7.3.2 Supply Type Spawn Points

Supply type spawn points are spawn points that provide supplies that can be picked up by the players in the game. Depending on the difficulty setting for the game supply spawn points might respawn an item a certain amount of time after it has been taken.

Spawning of Players

When placing players in the game world at the start of a game the game logic will lookup an available Player type **SpawnPointNode** for each player and place the player there. The availability of a **SpawnPointNode** is checked by testing whether a **BoundingCircle** with the same radius as the player's will collide with any dynamic objects when placed at the position of the **SpawnPointNode**.

Spawning of Zombies

Spawning and despawning of zombies is handled by the **ZombieController**. Zombies are spawned in the same way as players, by finding an available Zombie type **SpawnPointNode** and placing the zombie there. If no available spawn point is found while trying to spawn a zombie the zombie will simply not be spawned. Also each tile is allowed to spawn a maximum of 10 zombies. Any attempts to spawn additional zombies after 10 zombies has been spawned in a tile will be ignored.

At the start of the game zombies are placed in the $5 * 5$ tile square area surrounding the players, excluding the tile the players themselves spawn in. The number of enemies spawned will equal 10 times the number of available tiles to spawn in. Each zombie will be placed in a random tile meaning that there might not necessarily be 10 zombies in each tile at the start of the game. The maximum number of zombies in one tile will however be 10 due to the limit of maximum spawning 10 zombies in each tile.

The **ZombieController** keeps a list of tiles in which zombies can be spawned. This list is initialized to the tiles that zombies are spawned in at the start of the game and then updated each time a player moves from one tile to another. The list will contain any tile in the $5 * 5$ tile square area surrounding each player, excluding any tile that contains a player. When a player moves from one tile to another all zombies are despawned from tiles that are removed from the list and up to 10 zombies are spawned in each new tile added to the list. The tiles that are removed from the list will also have their spawn counter reset so that they can spawn another 10 zombies next time a player get close to them. In addition to spawning zombies whenever a player changes tile the **ZombieController** will also try to spawn a zombie in a random tile every two seconds.

Spawning of Big Zombies

Whenever the **ZombieController** tries to spawn a zombie in a tile it will check if there is an available BigZombie type **SpawnPointNode** in the tile. If that is the case a Big Zombie will be spawned instead of a normal zombie.

Other types of spawning

All other types of spawn points have a chance of placing an item into the game. Each **SpawnPointNode** has a spawn likelihood that can be set in the editor. This is a number between zero and one which represents the percentage chance that the spawn point will be active in a game. At the start of the game all spawn points are checked and a random number between zero and one is compared to the likelihood. If the number is greater than the likelihood the spawn point will be disabled for the current game. Otherwise the spawn point will be initialized. ZombieLauncher spawn points are special in that they when initialized simply place a zombie cannon in the game world and does nothing more for the rest of the game.

Initializing of supply type spawn points

When supply type spawn points are initialized they will have a model associated with them, representing the item that can be picked up from the spawn point. For health spawn points the amount of health that can be received from the spawn point is set and for ammo spawn points the number of clips to be received is set. Weapon spawn points will have the type of weapon set to one of the three types of small weapons. BigAmmo and BigWeapon spawn points are not used as these items are only available through dynamically generated spawn points in the game.

Random spawn points

A random spawn point has a certain chance of being converted to other types of spawn points. When random spawn points are encountered during the initialization of spawn points there is a one percent chance that the spawn point will be converted to a BigZombie spawn point, a 10% chance that it will become a Health spawn point, a 5% chance that it will become a Zombie spawn point and a 5% chance that it will become a Ammunition spawn point. The remaining 79% of the random spawn points will simply not be used.

Picking up items from spawn points

Whenever a player collides with a spawn point the game will try to give the item in the spawn point to the player. This is done by removing the item from the spawn point and trying to give it it to the player. If the player cannot receive the item the item is put back into the spawn point. When picking up ammunition or health the amount that can be picked up is simply added to the players current amount of health or clips. The player will automatically pick up weapons if he has room for the given type of weapon. Otherwise the player will have to press the pick up weapon button which will make him pick up the new weapon and drop his old one.

7.3.3 Respawning of items

When a player picks up the item in a spawn point, the spawn point will start counting down to a respawn of the object contained in the spawn point. The respawn time for each type of item is dependent on the difficulty selected for the current game. While the respawn counter is counting down objects cannot be obtained from the spawn point. It is possible to configure spawn points to never respawn items in which case the spawn point will provide its item only once. While respawning a SpawnPointNode will display a power-up timer model instead of the model for the item in the spawn point. Spawn points that never respawn will show no respawn model. The models used for the spawn point timers can be seen in figure 7.16 on page 81.

7.3.4 Dynamically generated spawn points

Some enemies drop items when they are killed. These items as well as weapons that are dropped when the player picks up a new weapon are placed in dynamically generated spawn points. These spawn points are special in that they provide one single item only once and are then removed from the game. If for example a player drops a weapon the weapon's object will be put into a spawn point and that exact same object will be given to any player that picks it up. This means that dropped weapons keep their ammunition count so that you cannot drop a weapon and pick it up again to gain additional ammunition.

When a dynamically generated spawn point has to be placed, several locations around the spawning object is tested. Each location is at a distance to the spawning object so that the new spawn point and the spawning object will not intersect. The locations is found by shooting rays out in 16 different directions around the spawning object and traveling the necessary distance along the ray. Each location is tested by checking if a BoundingCircle with the same radius as the SpawnPointNode placed at the location will intersect with any objects. If a location with no intersections is found then that location will be used for the new spawn point. If no possible location is found the spawn point will simply be created at a position offset 10 units along the X-axis from the spawning object's position.

7.4 Movement and Physics

This section briefly outlines the algorithms used for moving the characters in the game.

For each frame the collision system in the game engine is used to handle collisions between enemies, players and static objects in the game world. Enemies and players are represented by circles in the game and the game world is represented as polygons.

Whenever a player or an enemy has crossed from one tile to another the object which represents them will be removed from the tile they are leaving and added to the tile they are entering, at the same time changing their position to be relative to the new tile.

7.4.1 Player movement

Player avatars will move according to input from the player. The player avatar will face a given direction which the player can change with two keys, one for rotating left and one for rotating right.

When the player holds down one of these keys the avatar will rotate in the specified direction with a set rotation speed. When the player presses the key for moving forward the avatar will start to accelerate in the direction it is facing. This acceleration will continue until a given maximum speed is reached, after which the avatar will move forward at a constant speed. When the key for moving forward is released the avatar will decelerate until it stops. The key for moving backwards works the same way except it accelerates in the opposite direction.

7.4.2 Enemy movement

Enemies always move at a set speed, however the direction they move in follows two patterns: One for when they are close to their targeted player and one for when they are just roaming.

When roaming the the enemy will shoot out rays to the front, left, right and back and determine in which direction it can walk the furthest distance without hitting a wall, then start turning and walking towards the collision point with that wall. Roaming enemies will repeat this procedure every two seconds.

When in the same tile as their targeted player enemies will turn and walk directly towards that player. In this mode enemies will update their target point every 0.1 seconds. Enemies turn at a set turning speed, always turning the shortest distance to their given target.

7.5 Tileset Development

A large part of the engine is the work flow which enables creation of tiles, from conception to using the tile, or indeed tileset, in the game. The following describes the work flow which gets a tileset from the creator's mind and into the game.

Because of the way the level generator works, there are significantly fewer crossings than the other types of tiles in the average level. This means that when building tilesets fewer crossing type tile assets than the other types are needed. For a reminder of the tile types refer to figure 5.2 on page 37.

Step 1: The Concept

To make a tileset consistent, one should consider the basic concept it is to be based upon. All of this is based essentially on the same principles that one would find in the creation of a film set. The setting, the look and the feel of the stage should be consistent and believable, and as such the concept thoroughly thought through.

In the case of ORZAM the concept is that of an underground laboratory, the home of a mad scientist. This means doomsday devices, Tesla coils, large generators to power the whole thing if the government would try and shut the place down, as well as objects with no obvious purpose.

Step 2: Sketching

The next step is to sketch out each and every tile which is to be made for the tileset. The reason for sketching the tiles out first and not just starting, is to allow the tileset creators to settle on which assets are needed, but also because it is simpler to discuss the levels on paper first, than when they have already been created and are working in the game.

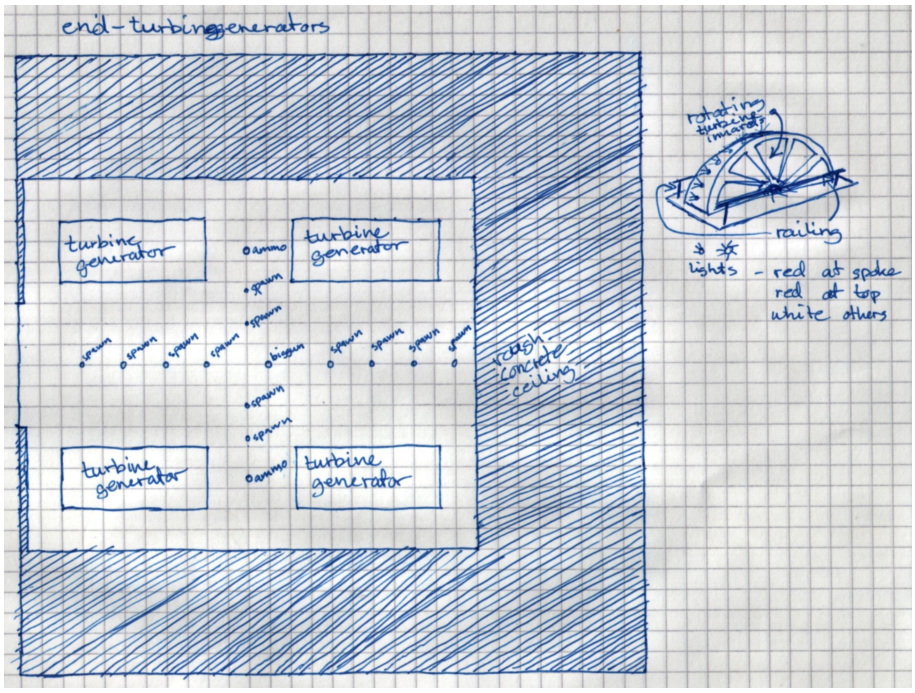


Figure 7.4: Sketch of the turbine generator room.

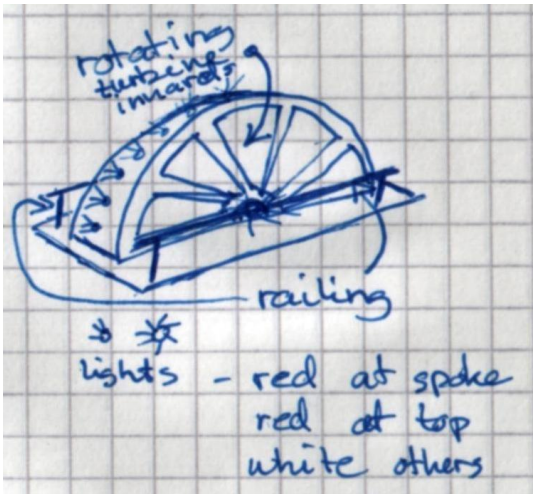


Figure 7.5: Closeup of the turbine generator asset sketch.

While testing the tiles out in the game is equally as important as the pre-implementation design that the sketching phase represents, and can force changes to the design (for example if a tile is shown to break the game flow when it is inserted in the game), it is important to discuss the design before the implementation of it is begun. This helps to streamline the look and the creation of the tiles, so that once that phase is begun, it can proceed at the highest possible pace, and the tiles, once in the game, can be tested for playability alone, rather than having to be tested for look as well.

The sketches used for creating the tiles in our game can be seen in appendix A on page 94.

To illustrate the process, a sketch has been completed in a more detailed form, as the sketches would look when creating a more serious implementation of such a game, and can be seen in figure 7.4 on the facing page.

Step 3: Creating Game Assets

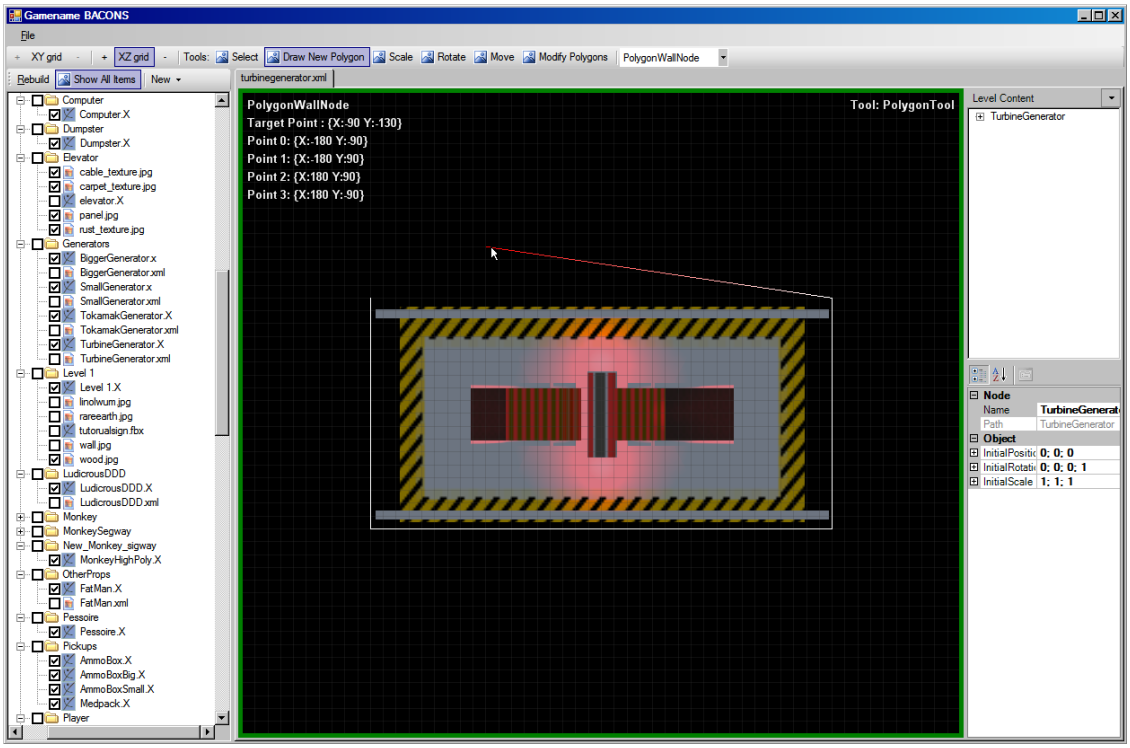


Figure 7.6: The turbine generator asset in the BACON editor, drawing the physics polygon. Note that this shows the use of the orthogonal camera in the editor.

A game asset is a scene graph created using the BACON editor. Creating a game asset is for convenience. A game asset is not the same as for example a 3D model asset, those assets are described in detail in the next section about asset work.

Lighting is done in the BACON editor, as lighting in EGGS is done differently than it is in 3D modeling packages. As described in section 5.5.1 on page 43, the lighting in EGGS is done through positioning point light sources in the scene, with a radius and an intensity. What this means from a level creator's point of view is that lighting a level in BACON is as simple as inserting and positioning point lights, and then assigning a radius and intensity.

To illustrate this process, we continue with the high-quality sketch seen in figure 7.4 on the facing page. In it, not only is the layout of the tile asset defined, but also a sketch of the turbine generator asset. A closeup of this area can be seen in figure 7.5 on the preceding page. This asset, once modeled by the 3D artist and exported to one of the supported formats, is handed over to the level creator, who puts the file into the content project, as described in the Engine Implementation chapter 5 on page 31.

The level creator then creates a new scene graph in the BACON editor, through `New→New Scene Graph`, which brings up an empty editing pane. For convenience, an **ObjectNode** is added into the level forest, though this is not strictly necessary - the usefulness of this, however, will become obvious in step four. The node, which for the time being is simply called "ObjectNode" is clicked upon and in the property grid, the Name value is changed to suit the asset which will be created. The asset file is then saved (`File→Save as...`) in the same folder as the model resides in, again for convenience, using the same name as was given to the object node.

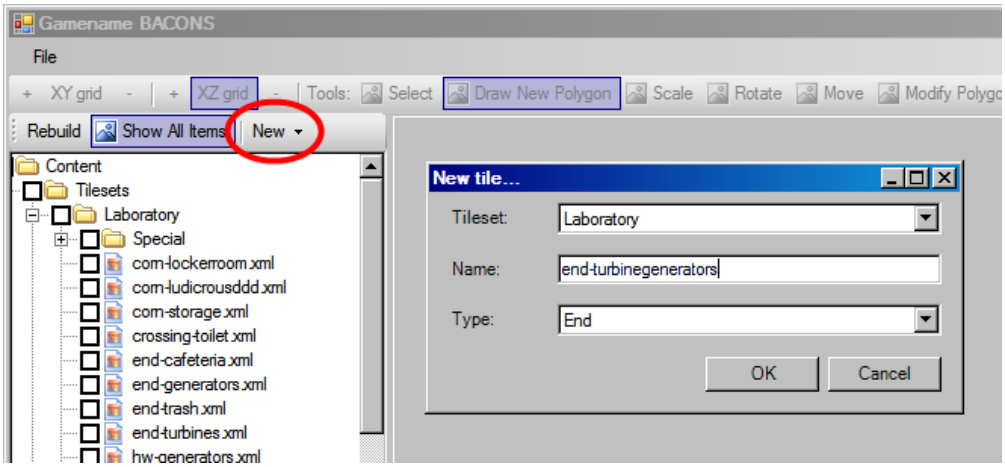


Figure 7.7: Creating a new tile in BACON - note the position of the New button.

The level creator now drags the model from the file list on the left and onto the object node. This will enable the model as part of the content project, and once dropped, the project is automatically rebuilt, the model loaded and shown in the view port. It may be required to change the scale of the model. It is important to change the scale of the model now, as changing the scale later on can become troublesome.

Right-clicking on the object node, the level creator chooses to add a new **PointLightNode**, which is then positioned using BACON's tools for manipulating position of objects (both direct input in the property grid below the level content, and using the Move tool). This is repeated for all the lights seen in the sketch of the turbine generator, adding the colour information to the three red lights as they are added.

The last item on the agenda for creating the asset is physics polygons. These define the areas of the model which will block the player in the level when the asset is inserted. This is drawn using the `Polygon` tool, with the **PhysPolyNode** selected in the drop-down box. Turning on the grid can be good for this purpose, as it helps to see where the model actually fits on that grid. The turbine generator is going to be used in such a way that a polygon around multiple generators would be disadvantageous, and as such, the physics polygon is drawn around the model in the asset rather than in the tile. Other objects, such as for example the Star Tracers (as seen in figure 7.21 on page 83) are used in such a way that drawing the physics polygon around multiple objects will be advantageous, and as such this step can be skipped.

Finally, after of course saving the asset with all the lights in place and the physics polygon drawn around the model, as it can be seen in figure 7.6 on the preceding page, the asset is closed by middle-clicking on the tab bar.

Step 4: Creating Tiles

Creating the actual tile in BACON is similar to how editing the assets functions, though with the addition of the wall polygon tools.

Step 4.1: Adding a Tileset

If this is the first tile of a tileset, a tileset must first be created. This is done by editing the Tilesets.xml file, found in the content folder under Tilesets. This file does not have a visual

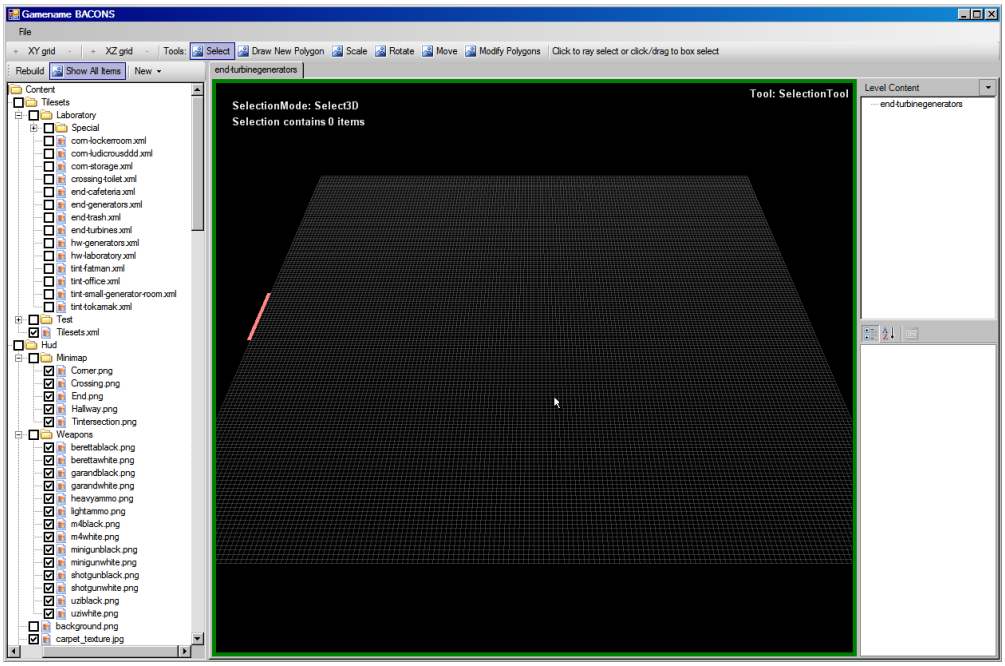


Figure 7.8: BACON with a newly created empty end tile - note that the editor area has been contrast-stretched, as while the grid is visible on a screen, it does not show well in screenshots on paper.

representation, but can still be opened in BACON , which then shows the list of available tilesets. Creating a new tileset is as simple as right-clicking on the empty area in the level content treeview and choosing to add a new **TilesetNode** which was described in section 5.3.1 on page 36.

Since the tileset in question, Laboratory, is already listed in the file, it will already be available, but after creating a new tileset, one should create a folder under the Content/Tilesets folder with the same name as the tileset (in this case Laboratory).

Step 4.2: Adding a Tile to a Tileset

The creation of tiles is where all the previous work is finally put together in a single, coherent structure, which eventually ends up in the game as a tile. When the game is first set up to use a tileset, it will automatically pick up any new tiles added to that tileset. More on that in the next step about using the tileset in the game.

To create the turbine generator tile, the level creator clicks `New→New Tile...`, selects the Laboratory tileset, chooses that the tile is of type End, and types in the name of the tile, seen in the sketch to be ”end-turbinegenerators”. The new tile is now created and BACON looks as can be seen in figure 7.8, ready to receive input. They now click `File→Save as...` to save the new tile, which is done in the tileset folder, which in this case is Content/Tilesets/Laboratory.

The first item of business is giving the tile a floor texture, which is done by selecting the automatically added TileInfoNode, which is named the same as the tile was named. Using the Texture drop-down to select one of the available textures, and the texture size and offset properties if appropriate, the texture is assigned.

Creating the walls in a tile is done in a similar manner to creating the physics polygons in an asset, and as such is done using the Polygon creation tool, though this time with **PolyWallNode** selected in the drop-down. Afterwards, an important extra step is to assign a height as well as top and wall textures to the new wall. The values of the last selected wall are remembered for the next wall, and as such the level creator will most often only set these values specifically once per type of wall they draw. Each of these wall nodes (here there is only one) are now given names and put into the **TileInfoNode** using drag-and-drop.

Once the basic geometry is done, the level creator drags the assets they created in the previous step and drops it onto the empty space in the level content tree-view. This copies the data from

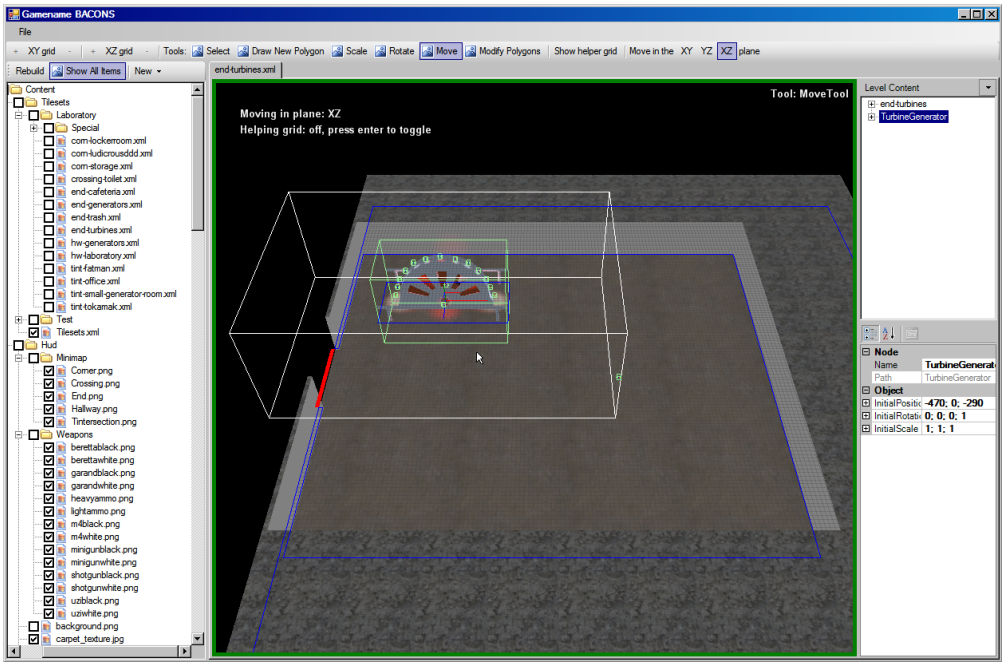


Figure 7.9: BACON with the first inserted asset selected, showing the automatic selection of all sub-items, as well as the white anchor of the top node.

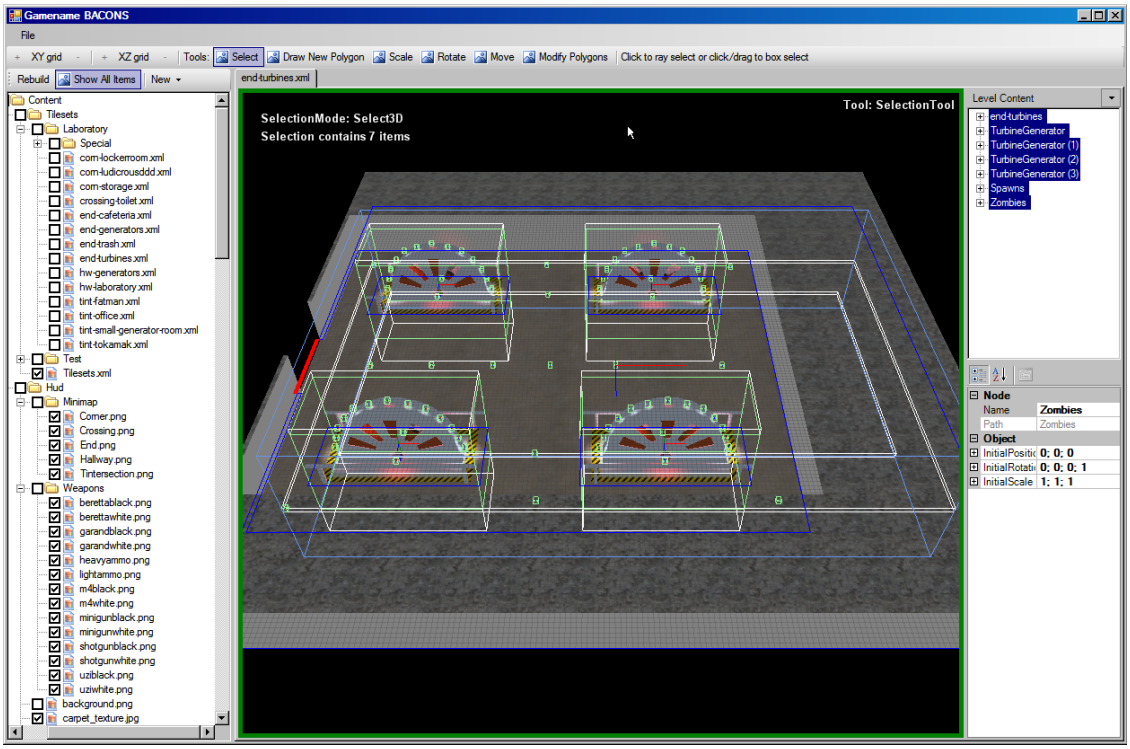


Figure 7.10: BACON with everything selected, showing bounding boxes and anchors for all nodes in the tile.

the asset into the level, and this is where it comes in handy, that everything in the asset was put under a single top node, as this node can now be moved around and the whole asset, including model, lights and physics polygon, is moved around in the scene in accordance (figure 7.9 on the preceding page). This is now used to position the asset in the scene, and repeated for all four turbine generators.

Any extra lights required in the tile can be added now, as all the geometry is in place. In this particular tile, no extra lighting is required, as the turbine generators have a lot of light on them already, and illuminate the tile fine on their own. Otherwise lighting is added in the same way as when creating assets - indeed, the lighting of specific assets can be changed as well, as they are copied data and not instances of the asset file.

Finally, once the entire tile is completed, **SpawnPointNodes** are added. This tells the engine where it is possible to spawn certain things, such as ammunition, weapons, zombies and so on. A special type, named **Random**, is selected by default, telling the engine that it is allowed to do anything it wishes with those points, but specific values can be set, and should be, when they make sense in the level. In this example, **SpawnPointNodes** of type **Ammunition** and **Weapon** are positioned in the points indicated on the sketch (figure 7.4 on page 72), and all other points are left as **Random**.

The tile is now completed, and the final result, with everything in the tile selected in the tree-view, is shown in figure 7.10 on the preceding page.

A Note on Special Tile Types: In ORZAM there are two special tiletypes: **Start** and **End** tiles. These are essentially normal tiles, however they are different in the way that they must be **End** tiles, they have the Special Type parameter set to either **Start** or **End**, and in the case of **Start** tiles, they have a number of **SpawnPointNodes** with type **Player**. Once the player enters an **End** tile in the game, the game registers that they have completed that level.

Finally, rather than putting them alongside the normal tiles, they are put into an extra sub-folder inside the tileset folder - in the case of the Laboratory tileset the folder Content/Tilesets/Laboratory/Special.

Step 5: Using The Tileset in The Game

In the game, the **TilesetHandler** class takes care of actually reading the tiles from disc, and assign them randomly to the **LevelGenerator**'s tiles. To use a specific tileset, the code in the game sets **TilesetHandler.CurrentTileset** to the name of the tileset, which should always be a value from the **TilesetHandler.AvailableTilesets** list. The following snippet shows the selection of the active tileset in the game:

Listing 7.2: GamePrototype/GUI/Screens/StatusScreen.cs

```
public static void LoadTileset(ContentManager content)
{
    TilesetWrapper wrapper = content.Load<TilesetWrapper>(TILESET_FILE);
    TilesetManager.TileSets.AddRange(XmlTest.LoadTest(wrapper.Tilesets).Where(x => x is TilesetNode).Cast<
        TilesetNode>());
    TilesetManager.Tiles = wrapper.Tiles;
    TilesetManager.SpecialTiles = wrapper.SpecialTiles;

    foreach (var item in TilesetManager.TileSets)
        if (item.Name == "Laboratory")
            TilesetManager.CurrentTileset = item;
}
```

Note that the current implementation is hardcoded to use the Laboratory tileset in all three levels, but this can be changed very easily, as this is the only place where the name is ever present in the code. Should the game programmer wish, it is trivial to change the tileset for a specific level.

7.6 Asset Work

In this section we will describe the different assets found in the game. Each of these were created in either 3D Studio Max or Softimage|XSI and then imported into the game engine. The assets

were created using a modeling technique called box modeling which simply is a way to extrude a polygon face and adjusting its size and position, and then repeat the steps until the model looks the way it was intended to. Some iterations of this method can be seen for the zombie bear model 7.11.

7.6.1 Character and power-ups

The main villain of the game is the Zombie Bear, this character is shown in figure 7.13 on the next page from its humble start as a primitive cylinder, to its present fear inspiring persona.

Figure 7.11: Showing the Zombie Bear from its start as a cylinder primitive.

Another enemy in the game is the cannon which shoots the morphing zombies at the player, it can be seen in figure 7.12 on the facing page.

Our hero the Segway driving, gun-wielding, radar capped, backpacking and spectacled monkey can be seen on figure 7.13 on the next page.

For the hero to have any hope of surviving the hordes of zombies, different kinds of power-ups were needed. These include health, ammo, and weapons. The game comes with a complete arsenal of weaponry, ranging from the pistol to the fully automatic M4 assault rifle, and topping it of is the Segway mounted minigun. Health and ammo packs can be seen in figure 7.14 on page 80, while a render of the weapons is seen in figure 7.15 on page 80.

Figure 7.12: The former Defence-O-Magic cannon which unfortunatly had been turned into an instrument of evil (now shooting the morphing zombie bears).

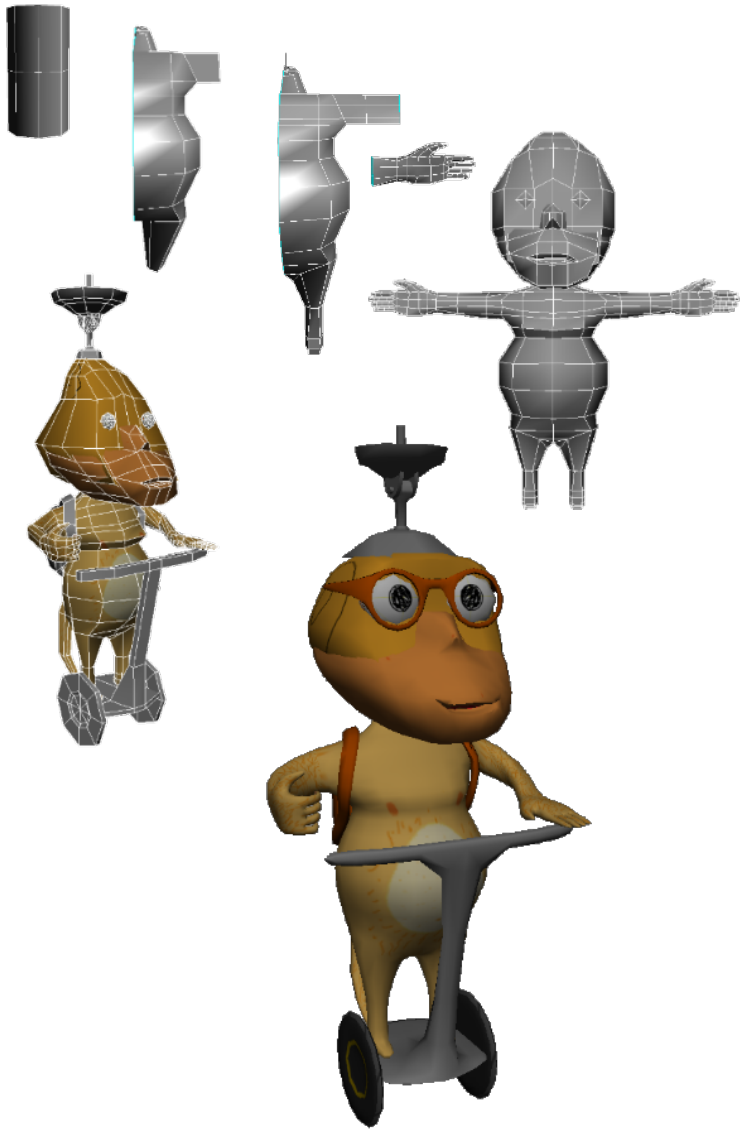


Figure 7.13: The Hero monkey on a Segway.

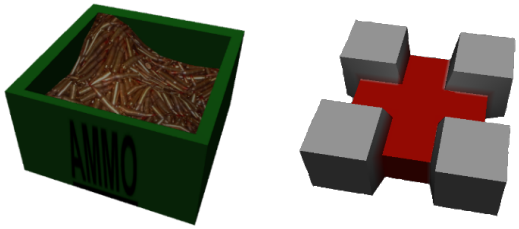


Figure 7.14: Ammo and a health box.

Figure 7.15: The weapons of the game. Left column: Beretta, Uzi and Shotgun. Right column: Garand, M4 and Minigun.

The power-up timers, as seen in figure 7.16, are displayed when an item is waiting to be respawned. The idea is to have three different models symbolizing the amount of time left. The first is the red countdown face, which is unhappy as it knows that there is still a long time before the item will reappear. This tells the player that there is no sense in lingering in the area just yet. The middle variant is the yellow discomfort face, which really does not know if it is worth waiting or moving on. Lastly we have the green happy face, telling the player that the power-up is about to spawn any second.

Figure 7.16: Red, yellow and green powerup timers.

7.6.2 Static assets

Multiple static assets were created and then imported into the game editor. To create a classic horror setting, the bathroom, a toilet is needed which can be seen on figure 7.17. We also need a dumpster, which can be seen on the right in the same figure.

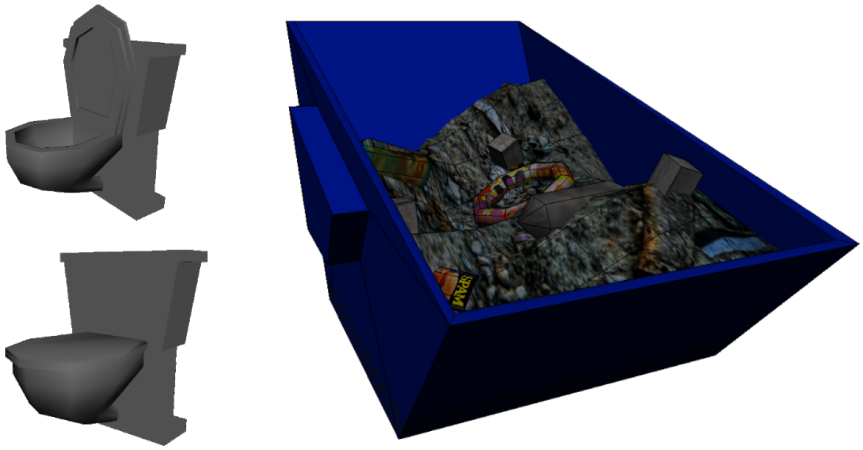


Figure 7.17: Showing the toilet with the lid in a up and downward position and a large dumpster with various garbage and Spam™.

In figure 7.18 on the following page we have the have the tables and a computer which combined can be use to create an office environment such as the cubicle hell seen in 6.2 on page 60. The tables alone can be used to create a cafeteria environment, as shown in the sketches in appendix A on page 94.

Every mad scientist has to have power and thus multiple kinds of power generators were made, some fictitious, other real. Mad scientists are friendly to the environment, so at least one of them has to be a turbine. Regular power generators will also do. Both types can be seen in figure 7.19 on the next page.

The Tokamak generator is still in a testing stage in reality, but this makes it an excellent asset for the game, as the scientist is not constrained by reality. All that power has to be used on something, thus the Tesla coil. Both the Tokamak generator and the Tesla coil can be seen in figure 7.20 on the following page.

With all the more normal assets done, it is time to show the mad scientist's true colors. This is done by placing a lot of doomsday devices in his lab. Of course he has to have a nuclear

Figure 7.18: The two tables in the game, one plain and one with a sink, and a computer showing Blue Screen of Death.

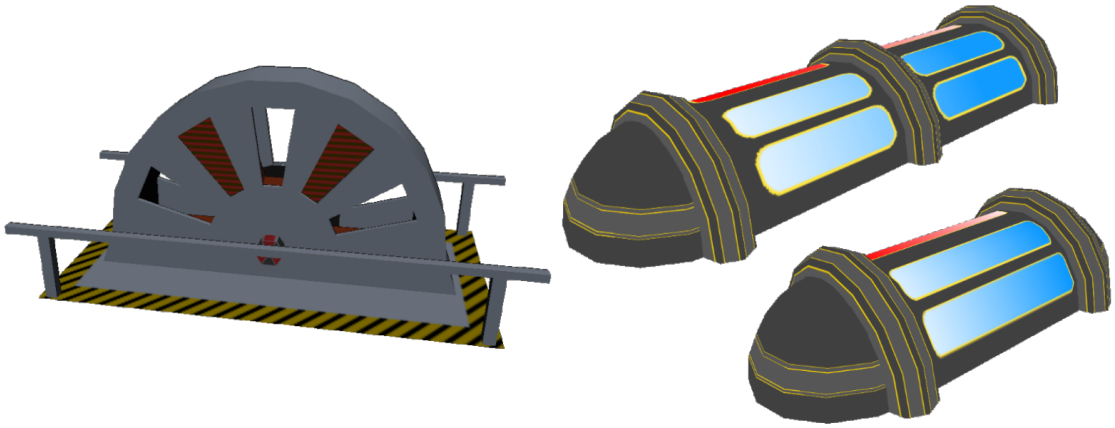


Figure 7.19: A turbine and a small and a medium power generator.

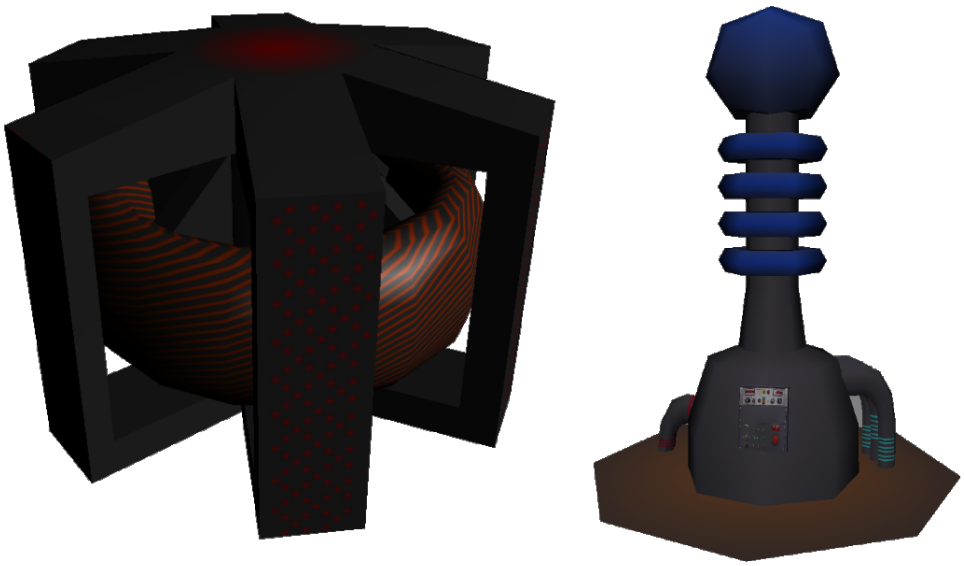


Figure 7.20: The Tokamak generator to the left, and a Tesla coil to the right.

bomb, seen to the right in figure 7.21. When the scientist is done with this world he can wreak destruction on nearby star systems with his Star Tracer device seen to the left in the figure. If that is not enough the Ludicrous Dooms Day Device, seen in figure 7.22, can dish out some doom on the rest of the doomed universe.

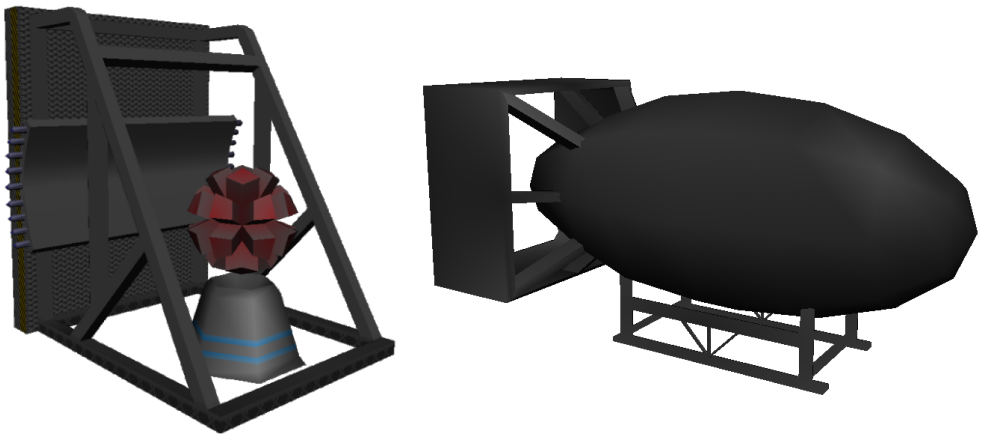


Figure 7.21: The Star Tracer and the Fat Boy.

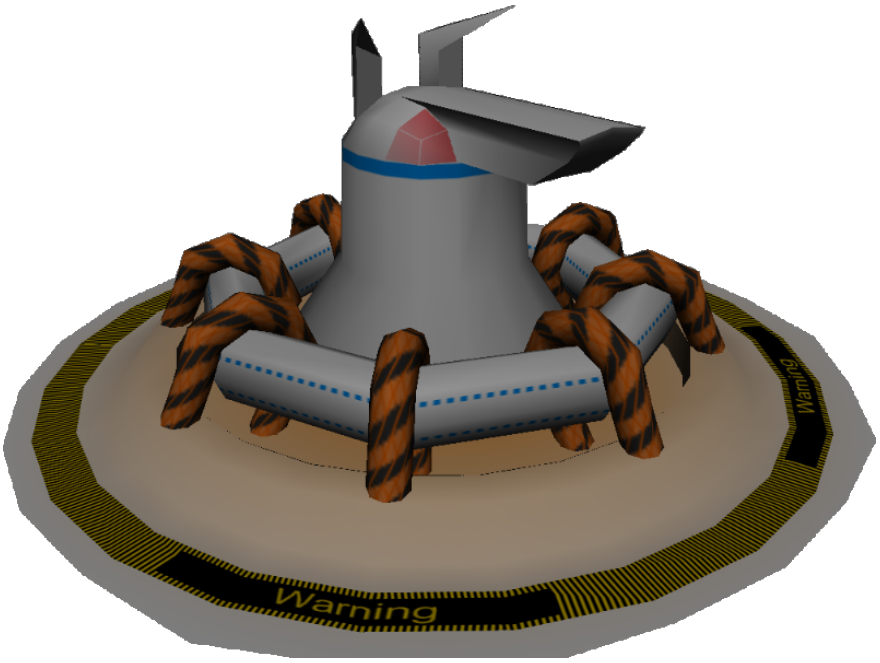


Figure 7.22: The Ludicrous Doomsday Device.

Last but not least, no game would be complete without the game classic, the crate. Crates like these have found their way into many games over time, among others can be noted Doom and Half-life. Thus they should not be left out of our game and consequently can be seen in figure 7.23 on the next page.

7.7 Special Effects

To spice up ORZAM some special effects have been added to the game. These special effects are described in the following sections.

Figure 7.23: Storage Crates (and maybe a box).

7.7.1 Particle System

A particle system is a system which emits and controls particles in the game world. Typically the particle system will allow the creation of emitters that can create particles at a certain point and give them a direction and speed. The direction and speed are often randomized so that objects are sprayed out from the emitting point. The particle system will update the emitted particles and move them according to the rules of the system. It will also make sure that particles are removed from the game world when they are no longer needed.

7.7.2 Blood Particles

Whenever an enemy is hit in the game the Blood Particles system will emit blood particles. The particles will originate from the point where the enemy's bounding circle is hit offset at a fixed height above the floor. The direction of the particles is based on the direction of the shot that hit. Each particle will have a jitter added to its direction so the particles are sprayed randomly. Blood particles will move in their designated direction, slowly changing the direction to point towards the floor over time, making the blood particles move in an arc from the emitting point and towards the floor. Whenever a blood particle hits the floor it will be removed from the system and for every 50 particles removed in this way a blood spatter will be drawn on the floor. The textures used for the blood spatter is generated using the Blood Spatter Generator. Blood particles are rendered as points.

7.7.3 Blood Spatter Generator

Since so many other items in ORZAM are procedurally generated, creating blood spatter textures in a graphics program seemed like the wrong way to go. So, a blood spatter texture generator was devised, inspired in part by the work by O'Brien and Hodgins[6] on computer simulated splashing fluids, which is based on creating dynamic simulations of fluid systems, including splash effects in pools of liquid.

The simplified approach used in the blood spatter generator is shown in listing C.3 on page 102, and some sample output can be seen in figure 7.24 on the next page.

7.7.4 Traces

For every five shots fired in the game a trace particle will be emitted from the fired weapon. Trace particles will move in the direction of the shot being fired and will be removed from the system once they have traveled the distance to the first object being hit by the shot. Traces are rendered as red line segments. In addition to rendering the line segment a red point light will also be placed at the position of each particle.

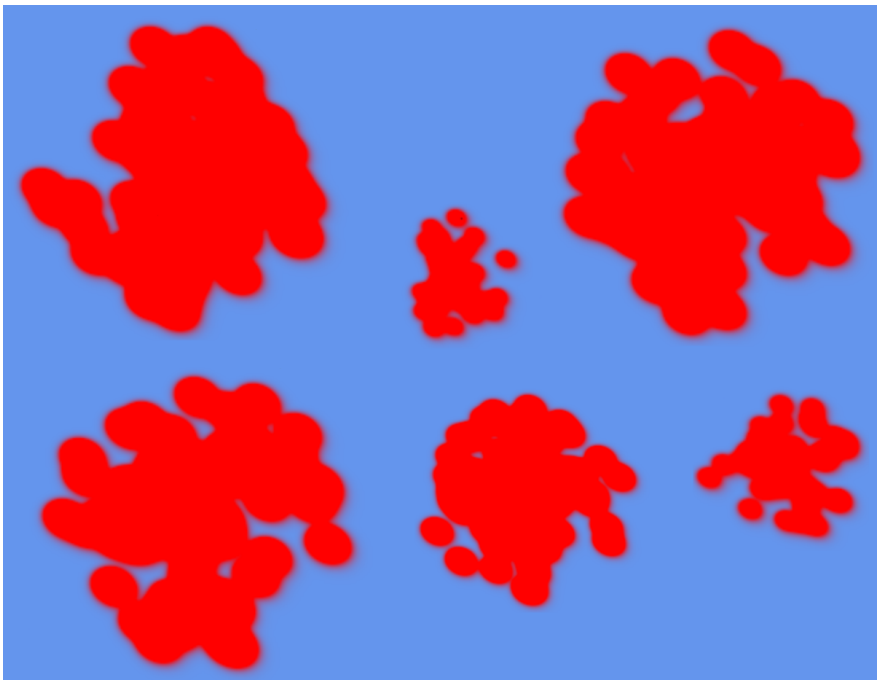


Figure 7.24: Various sample outputs from the blood spatter generator.

7.7.5 Sparks

Sparks are emitted whenever a shot hits a wall or a zombie cannon. Their direction will be found by reflecting the direction of the shot around the normal to the object being hit and then adding some jitter. Like blood particles sparks will slowly change their direction to point towards the floor. Sparks will be removed after having existed a certain amount of time or whenever they hit the floor. Sparks are rendered as wheat-colored line segments and will have a wheat-colored point light placed at their position.

7.8 Game State Implementation

The section describes the graphical user interface (GUI) implementation of ORZAM and how it is used to manage game states. The implementation is built using the engine Screen System, which is described in section 5.8 on page 57.

XNA provides support for easily drawing bitmaps to the screen. This also includes sprite fonts for drawing text strings. However, XNA does not provide any high-level tools for building GUIs, so implementing GUIs does require quite a few lines of code.

The thoughts behind the GUI were to keep it simple. We only wanted a GUI, which has the most basic functionality, for example starting a game, setting up controls, the number of players and sufficient functionality to support the gameplay.

7.8.1 Class Organization

The GUI-implementation is spread across a number of classes, which all extend the abstract GameScreen base class. The following classes are implemented:

MainMenuScreen.cs

A rather simple class, which draws a string array containing menu options centered on the screen. Menu navigation is done by manipulating an index, which keeps track of the currently selected menu option, which is displayed in a different color than the other menu option.

The class can be seen in use as **(1)** in figure 7.25 on page 88.

ConfigureControlsScreen.cs

This class provides functionality for setting up keyboard controls for two players.

All keyboard controls are represented in a class which is called **PlayerControl**. It contains arrays of keyboard keys for the players and a corresponding string array with in-game descriptions of the key functions.

The ConfigureControlsScreen class can load and save a PlayerControl object, by using the binary serialization feature of .NET. The ConfigureControlsScreen class provides a visual representation of a PlayerControl object by drawing it to the screen. It also provides functionality for receiving input and changing the keyboard key values in the PlayerControl object.

The class can be seen in use as **(3)** in figure 7.25 on page 88.

DialogueScreen.cs

This class is a simple pop-up dialogue, which can be used in two ways:

1. Display an array of menu options and return the name of the menu option being selected by a user. This can be seen on the pictures **(2, 4, 5, 8)** in figure 7.25 on page 88.
2. Display a string and return which keyboard key the user presses as a string. This functionality is used in conjunction with the ConfigureControlsScreen class, where the user is presented with dialogs containing text such as "Press a new key for player 1 fire". This can be seen in **(6)** in figure 7.25 on page 88.

Contrary to the other GUI classes, the DialogueScreen is intended to be rendered on top of another screen, instead of being rendered as full-screen. In practice this is done by exposing XNA's GraphicsDevice class' ability to draw only to a specific region of the screen surface.

SetupGameScreen.cs

The SetupGameScreen class contains functionality for a primitive "game lobby" for up to 4 players, using up to two different keyboard layouts or up to four gamepads. It takes a very simple approach to selecting the number of players and which in-game controls they use. Players simply press their "fire key" in order to join the game. The advantage of this approach is that it is simple to handle multiple input devices.

The class can be seen in use as **(7)** in figure 7.25 on page 88.

StatusScreen.cs

This class has a higher coupling to the game logic, as opposed to the other GUI classes, which for the most part is only used as presentation layer for the game logic.

StatusScreen has an instance of LevelGenerator (which is described in section 5.4 on page 38) and it also takes a few parameters: The number of joined players, the chosen difficulty and the current level. These parameters are used for generating a level and displaying the correct "level story".

The class also displays a visual overview of the generated level, which serves as a help to the players. The visual overview suggests two possible exits on the generated level. One of the suggestions is correct and the other is misinformation. The point is to only provide a minor help, in order to increase game difficulty.

The class can be seen in use as **(10)** in figure 7.25 on page 88. The visual level representation is topmost and the level story is in the center.

InGameScreen.cs

This class contains our in-game logic, which is encapsulated in the GameWrapper class. So this class is quite simple, because it mainly just calls the Update() and Draw() methods of the GameWrapper.

However, InGameScreen does also contain functionality for an in-game dialog with the options "Exit Game" and "Resume Game". It can be seen in figure 7.25 on the following page: InGameScreen as **(9)** and InGameScreen with dialog as **(8)**.

7.8.2 Menu Navigation

This section describes the navigation in figure 7.25 on the next page.

From the main menu **(1)** there are three options:

1. It is possible to quit the game, which will present a "Really Exit?" dialog **(2)**.
2. It is also possible to change keyboard controls **(3)** **(6)**, which has a feature for reminding the user, if he changed something and did not save it before trying to return to the main menu **(5)**.
3. Finally is it possible to start the game. First the user is presented a dialog where game difficulty must be selected **(4)**.

After selecting game difficulty a simple "game lobby" is presented, where players can join by pressing the "fire key" of whichever input device they are using.

With one or more joined players, the game can proceed. A "status screen" with informations for the first level is displayed **(10)**. This consists of a text with a small story and a picture, which gives an outline of the level, with starting position and two possible exits.

Pressing Enter on a keyboard or Start on a gamepad makes the game continue to the actual in-game **(9)**. Surviving a level and finding the exit will result in a "level statistics" screen **(11)**, presenting various kill and damage statistics. Dying will result in a screen telling the player or players to try again **(12)**. Further navigation from both **(11)** and **(12)** will lead back to the status screen, but only finding the exit will result in the status screen changing to the next level.

7.9 Audio System

The audio system in ORZAM is split into two parts - the generic sound effects and music playing class **AudioManager** and the game-specific **BearSoundsEmitter**.

7.9.1 AudioManager

We implemented an AudioManager class because we experienced some crashes on machines without sound devices. The AudioManager provides a simple interface to playing sounds, while hiding any kind of error handling.

7.9.2 Bear Sounds Emitter

We created the BearSoundsEmitter class give the game a scary ambiance in the game. In short, the class launches a quiet, random vicious bear sound, every one to three seconds, at a random position around the player. It does however use advanced 3D audio positioning technology for maximum effect.

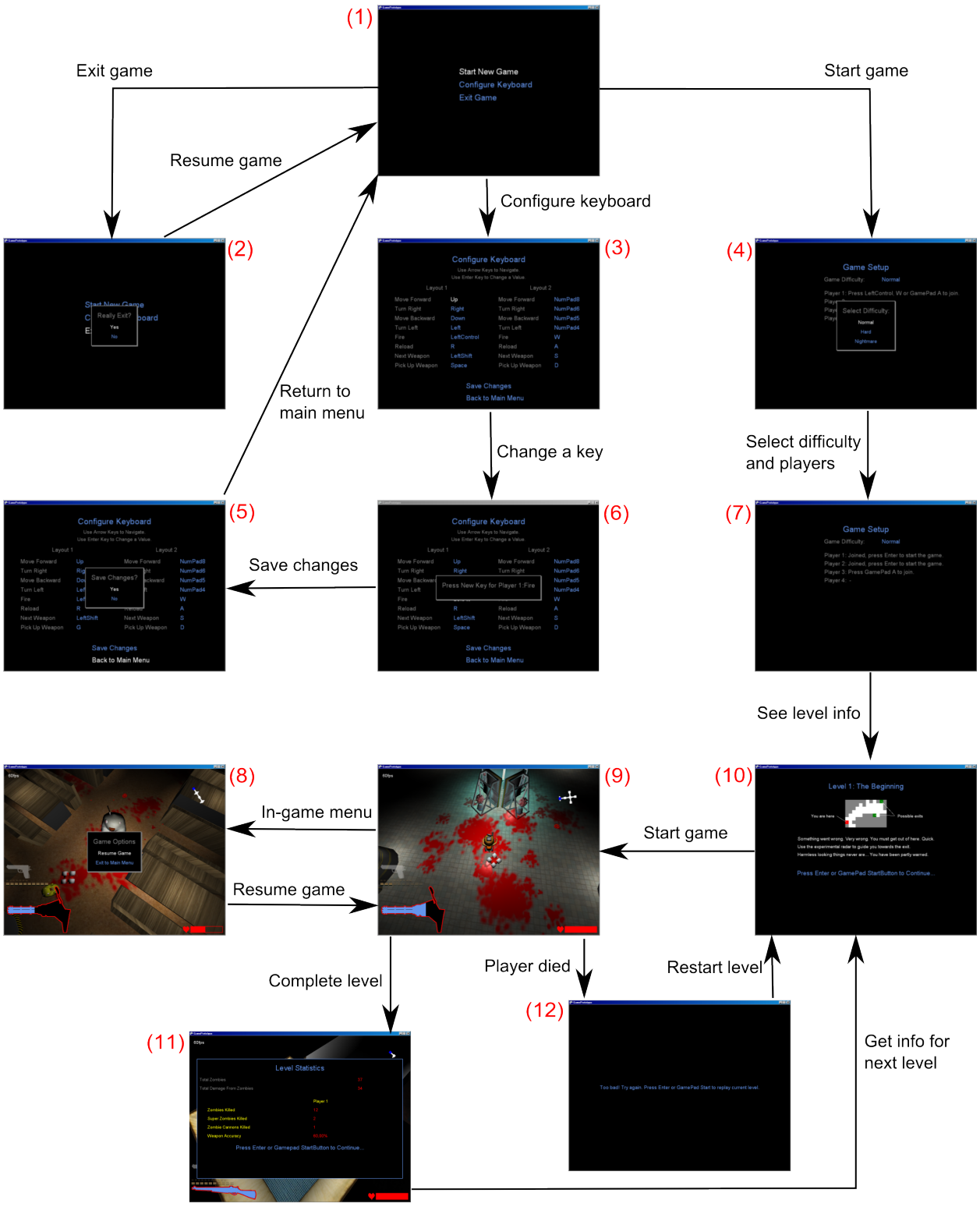


Figure 7.25: Graphical User Interface navigation.

CHAPTER

8

Conclusion

8.1 EGGS Engine as a Development Platform

This section evaluates the usage of the developed engine. We highlight strengths, weaknesses and complications encountered during the development of the ORZAM game.

8.1.1 Specialized Engine

Because we built the engine with the purpose of creating the ORZAM game, it has features that are specialized for creating this kind of game. The BACON editor was a big help in creating content quickly and correctly.

The engine is designed for creating 2.5D games and should therefore be very reusable for these kinds of games. However, if we were to use the game for creating a purely 3D game, parts of the engine would need refactoring and several 3D specific features would have to be implemented from scratch, for example 3D collision response.

8.1.2 Software Development Kit (The BACON editor)

Using the BACON editor is a quick way of building new game worlds, however because of its short development cycle it is not complete. It lacks features such as undo and it has a tendency of being somewhat unstable.

The implementation of scene graphs in the engine is very robust, such that changing or adding node types does not invalidate previously created scene graphs. Also when adding new node types they are automatically recognized by the BACON editor. Finally, the file representation is XML and therefore easy to debug and understand.

8.1.3 Performance

The engine performs very badly on machines with low-end graphics devices. More optimization is needed in the renderer of the engine. Also, guidelines are needed for content creators in order to avoid slowdowns caused by too complex content.

The collision system in the engine is easy to use, but it has some problems when the game is running slowly. Because the physics of the engine is time based, it has trouble with missed collisions when the time steps become too large due to low framerates.

8.1.4 Content Support

The engine can handle a wide variety of different content, including skinned and animated 3D models.

However, when the amount of content increases so does the time it takes to build the game and levels. The editor's content management system is not entirely compatible with Visual Studio's build engine. Even if the content has been built through the editor, Visual Studio will rebuild all content when building the game. The same goes for the other way around.

8.1.5 Final Thoughts

While the engine is not production ready, it can easily be used to implement visually compelling games. Because it is based on XNA it is also ready to be used for an Xbox 360 game, and is well suited for this because of its support of spilt-screen gameplay and gamepad control schemes.

8.2 Conclusion

In the introduction 1 on page 8 we formulated a goal for this project. In this section we will account for whether this goal was reached. The project goal was the following:

> *We want to get hands-on experience with game engines by implementing a game using a game engine.*

We have achieved this goal by creating our own game engine and used it to implement a fully functional game, based on our initial game design document as specified in chapter 2 on page 9.

By developing our own game engine we have amassed hands-on experience with the inner workings of game engines.

We consider involving content creators in the development of a game very important, which is why we developed and used a Software Development Kit for this project. By doing so we believe we followed a more realistic game development process.

8.3 Future Work

In this section we will describe our plans for future development of the engine and its SDK.

8.3.1 Further Optimization

Optimizing a rendering pipeline can be tricky. By not updating objects that are far away we optimize the Application Stage, by not drawing objects that are outside the camera's view frustum we decrease the workload in the Geometry Stage, but the game is still slow on machines with low-end graphics capabilities. This why we suspect the rasterizer of being overburdened.

As stated in Real-Time Rendering[1], this is easily verifiable, because the amount of work the rasterizer needs to do is directly proportional with the amount of pixels it needs to shade. This means that if lowering the screen resolution of the game increases the amount of rendered frames per second, our bottleneck is in the Rasterizer Stage.

We ran a couple of tests using different resolutions on a machine with an "ATI Radeon Mobility X1800" graphics device, which supports Multiple Render Targets. Using 1024x768 the game ran at 35 frames per second. Decreasing it to 640x480 resulted in 44 frames per seconds, and finally a resolution of 320x240 yielded 61 frames per second.

Obviously the bottleneck of our game is in the rasterizer.

The reason our rasterizer is so heavily burdened is because of Deferred Shading. While the technique has very nice properties, the main drawback of the technique is that it puts a lot of load on the rasterizer, especially if the graphics device used to render the scene does not support Multiple Render Targets very well. Some older graphics devices that do not support MRTs simply run the pixel shader once per render target, which triples the workload of the rasterizer.

Also, our deferred shading algorithm uses per-pixel lighting exclusively with no fallback to per-vertex lighting. To support more platforms we should develop fallback shading techniques.

Finally, the instruction count of the pixel shaders in our deferred shader implementation is high. Some effort could be put into reducing the amount of instructions for example by using less precise approximations.

8.3.2 Collision Detection Quirks

As stated in section 8.1 on the preceding page, the collision system does not handle low framerate situations very well. This could be improved upon by either using fixed time steps or implementing a more advanced time stepping algorithm.

8.3.3 Product Polish

The engine and the editor could use more polish. That is more time spent on fine tuning features and squashing bugs. One important example is adding an undo feature to the editor. Another example is thoroughly testing the Xbox implementation of the engine. Finally, some documentation would also be nice.

[1] Tomas Akenine-Möller and Eric Haines. Real-Time Rendering. A K Peters, second edition, 2002. 16, 17, 19, 20, 36, 46, 90

[2] Garret Foster at gamedev.net. Scene graph. http://www.gamedev.net/reference/programming/features/scenegraph/. 14

[3] David Eberly. Intersection of linear and circular components in 2d, 2000. http://www.geometrictools.com/Documentation/IntersectionLine2Circle2.pdf. 48, 50

[4] Shawn Hargreaves. Deferred shading, 2004. http://www.talula.demon.co.uk/DeferredShading.pdf. 43, 44

[5] Microsoft. Xna creator's club. http://creators.xna.com/. 42

[6] James F. O'Brien and Jessica K. Hodgins. Dynamic simulation of splashing fluids. In Computer Animation '95, pages 198–205, Atlanta, CA 30332-0280, April 1995. Georgia Institute of Technology. 84

[7] James M. Van Verth and Lars M. Bishop. Essential Mathematics for Games & Interactive Applications. Morgan Kaufman, second edition, 2008. 50

[8] Jon Watte. Kilowatt x-port. http://www.kwxport.org/. 52

[9] Wikipedia. Scene graph. http://en.wikipedia.org/w/index.php?title=Scene_graph\&oldid=244200368. 14

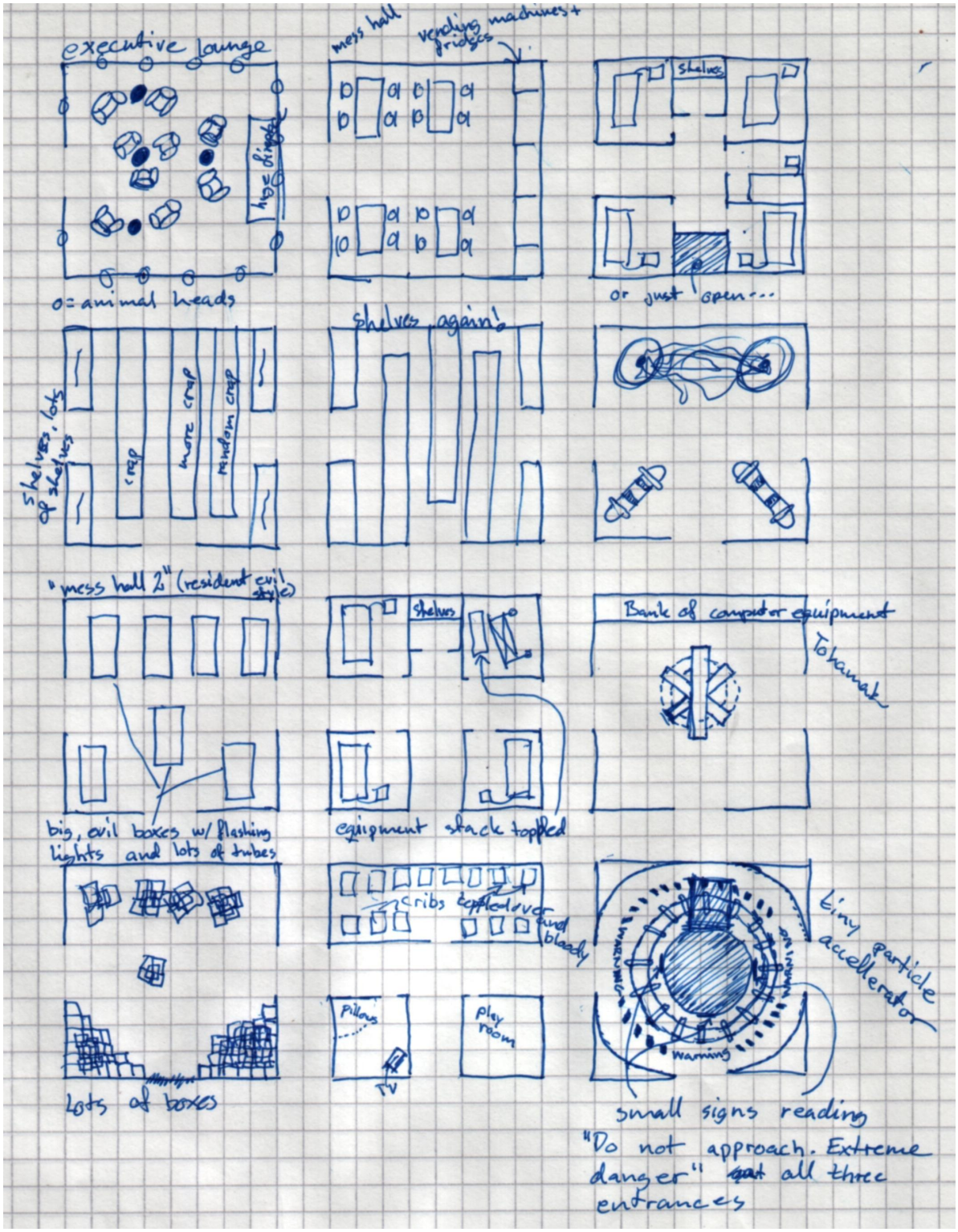
[10] Catalin Zima. Deferred shading in xna, 2007. http://www.ziggyware.com/readarticle.php?article_id=155. 42, 43

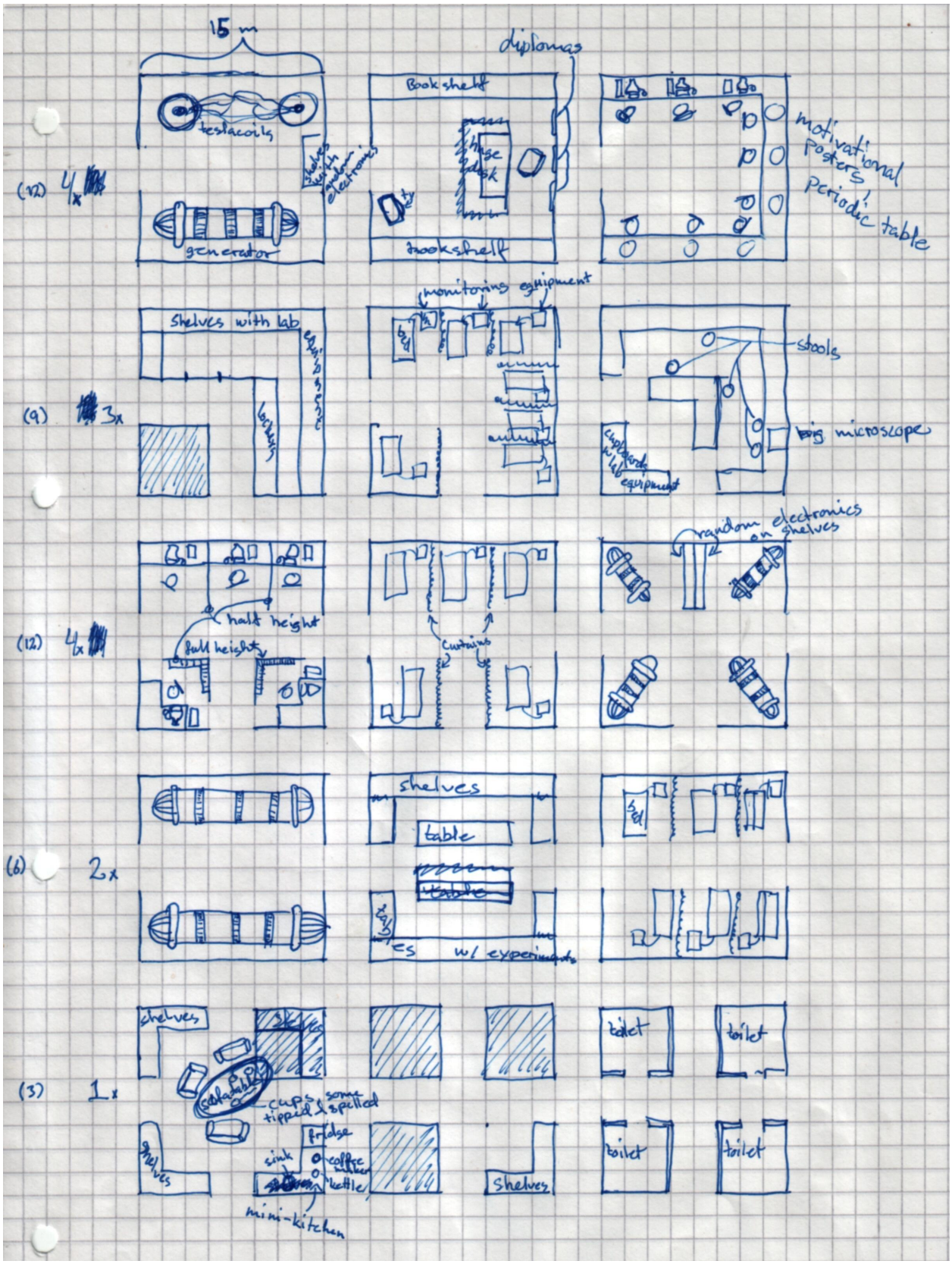
Part I

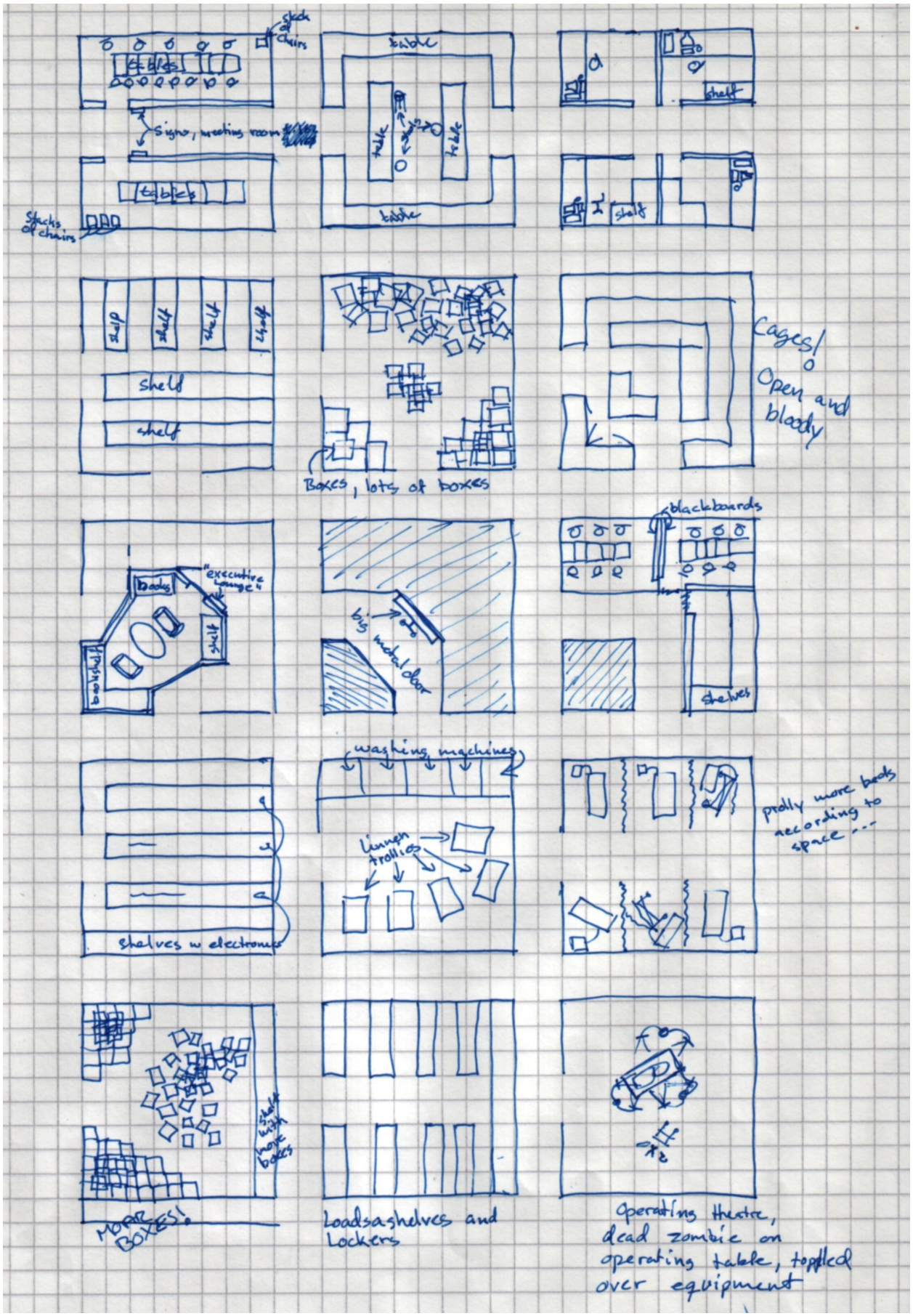
Appendix

Sketches

The next three pages contain the sketches of the tiles used in the game.




15 m
diplomas
Bookshelf
testacoils
generator
bookshelf
motivational Posters / periodic table
(12) 4x
monitoring equipment
shelves with lab equipment
lockers
stools
big microscope
(9) 3x
random electronics on shelves
half height
full height
curtains
(12) 4x
shelves
table
bed
(6) 2x
shelves
toilet
toilet
(3) 1x
cups & some spilled
fridge
sink
coffee maker
kettle
mini-kitchen
shelves
toilet
toilet


Stacks of chairs
Signs, waiting room
tables
table
table
table
shelf
shelf
Stacks of chairs
shelf
shelf
shelf
shelf
shelf
shelf
Boxes, lots of boxes
Cages!
Open and bloody
"executive lounge"
books
shelf
shelf
big metal door
blackboards
shelves
washing machines
linnen trollies
pretty more books according to space...
shelves w electronics
Loads of shelves and Lockers
operating theatre, dead zombie on operating table, toppled over equipment

Article: Write Games, Not Engines

The following article is from the blog at `http://scientificninja.com`. The author write the following in his "About" section at the webpage:

My name is Josh Petrie. I am a tools programmer in the game development industry. At some point in the future, a pithy "about the author" page will go here. Until such time as I find somebody who I can bribe to write suitably wonderful things about me, you'll have to make do with this sparse substitute. Enjoy.

Write Games, Not Engines

To begin with, the term "engine" (specifically as it related to the game development world) has no strict definition. Therefore, in the interests of keeping everybody on the same page, I'll define the term as I intend to use it in this article. An "engine" is a collection of robust, reusable software subsystems (possibly including both code libraries and tools) designed to facilitate the development of actual games by addressing specific requirements. The requirements tend to be broadly-defined: rendering, audio, physics, et cetera. Particularly ambitious engines that address multiple broad requirement groups tend be to known as "game engines" rather than just "graphics engines" or "physics engines".

Now that that's out of the way, let's turn to the real issue: how to build engines, and more specifically, if you should even bother.

For any number of reasons, many neophyte game developers (and even some moderately experienced ones) seem to think that a game engine is a required, critical part of a game. They think that – in keeping with the mechanical analogy from which the term "engine" comes from in the first place – like a car, a game with no engine will simply not run. But that mechanical analogy starts to break down when you take it this far, because when you look at the reality of the situation, a game engine is about *reusable* components. Much like you can build a car with custom parts or generic ones, you can build a game with custom components or reusable ones. The fact that you must have an engine to build a non-trivial game is a fallacy, something perpetuated largely by people who don't know any better.

Nonetheless, every so often comes the obligatory post on GDNet (A forum for game developers, red.) by somebody who has decided its time for them to write *their* engine, so they can start making some *real* games. These posts usually consist of requests for resources about the engine design and development process, and in fact chances are that if you're reading this, it's because I caught you making just such a post and linked you here so that I could crush your dreams with a minimum of effort on my part.

So my advice to you, if you're trying to write an engine, is: *Don't.* No matter what your reasons are – it doesn't matter if you're writing an engine so you can write your dream game, or if you're writing an engine because you think it will be a good learning experience, or any number of similar reasons. They're all wastes of time. You can sit down and write a game without writing a pre-written engine, and in fact this is very often the better approach, regardless of why you want to write an engine. The entire development process goes *much* more smoothly if you are focused on writing a *game* instead: a game is much easier to identify requirements for, much narrower in focused, much more rewarding when finished, and much, much more useful.

Most hobby developers who "finish" an "engine" that was designed and built in isolation (with the goal of having an engine, not a game, upon completion) can't ever actually use it, and neither can anybody else. Since the project didn't have any goals, any boundaries, or any tangible applications, it essentially attempted to solve aspect of the chosen problem space and consequently

failed miserably. These "engines" tend to be little more than an inconsistent and messy amalgamation of random functionality that isn't terribly practical, robust or reusable. Furthermore, the actual viewable end product of these "engines" tend to be a cute little map (loaded from one of the popular BSP formats) you can run around in and observe some cute little models (also loaded from a popular model format) rendered with some fairly basic lighting effects (maybe some per-pixel normal mapping or parallax mapping). Maybe you might even see some collision detection or animation. These projects are just tech demos, and not particularly compelling ones at that. There is nothing to them, and they feel rather flimsy and inflexible.

The solution, even if you *really* want to make an engine, is to make a *game* instead. I can hear you crying out in protest from here, but just bear with me for a minute. The game does not have to be an epic production. It does not have to be GTA5, Quake 17, the next Elder Scrolls game, or a WoW-killer. It just has to be a *game*, with well-defined (and ideally well-thought-out) gameplay and well-defined developmental scope. A game is much easier to identify requirements for, as I've already mentioned, and it is also a practical application of those requirements. Once you have made one game, make another. Then another. Each time you start a new project, you should identify functionality that could be used and pull it out into a common library of base code. You'll probably have to refactor some of your code to remove explicit dependencies on other code or data that is game specific, but this is a *good* thing. It will help you generalize what is generalizable in a way that you can still test the generalized functionality against your finished game and confirm that it still works (obviously you might have to modify some game-specific code to adapt to the increased generality, as well).

If you repeat this process long enough, after a few games you'll have the beginnings of a solid collection of reusable functionality that has been *proven* to have practical applications. You'll also have a set of far more interesting demonstration applications that could also double as test harnesses for your engine. This method of growing an engine (rather than manufacturing it from whole cloth) is superior because it helps you shape your goals, it forces you to actually *think* about the problems you will face and their practical implementations, it forces you to think about the separation between the domain-specific logic of the engine and the game. It makes you learn from your mistakes rather than pretend that mistakes cannot be made.

Code Examples

C.1 Node Reflector Class

Listing C.1: Loading a Scene Graph from an XmlDocument

```
1  // This is the function we call to load a scene graph
2  public static NodeList LoadScene(XmlDocument loadDoc)
3  {
4      NodeList nodes = new NodeList(null);
5      LoadRecursively(nodes, loadDoc.DocumentElement.ChildNodes);
6      return nodes;
7  }
8
9  public static void LoadRecursively(NodeList targetList, XmlNodeList xmlNodes)
10 {
11     foreach (XmlNode xmlNode in xmlNodes)
12     {
13         XmlElement element = xmlNode as XmlElement;
14
15         // Not an element or doesn't have a Name attribute, then ignore this element
16         if (element == null || !element.HasAttribute("Name"))
17             continue;
18
19         SceneNode node;
20
21         // If we cant instantiate the node, then continue to next xml element
22         if (!NodeReflector.TryInstantiate(element.Name, element.Attributes["Name"].Value, out node))
23             continue;
24
25         // Set the Properties of this node from using the XML
26         node.SetFromXml(element);
27
28         // Put it in our list
29         targetList.Add(node);
30     }
31 }
32
33 public static class NodeReflector
34 {
35     public static bool TryInstantiate(string type, string name, out SceneNode n)
36     {
37         n = null;
38         foreach (var t in NodeTypes)
39         {
40             if (t.Name != name)
41                 continue;
42
43             // Get a constructor on the type that takes one string argument, and then invoke it with name as
                   the argument
44             n = t.GetConstructor(new Type[] { typeof(string) }).Invoke(new object[] { name }) as SceneNode;
45         }
46         return n != null;
47     }
48
49     // Ensures that we only find the types we need once (singleton-ish)
50     public static List<Type> NodeTypes
51     {
52         get
53         {
54             if (nodeTypes == null)
55                 FillTypeList();
56
57             return nodeTypes;
58         }
59     }
60     private static List<Type> nodeTypes = null;
61
62     // Runs through all loaded assemblies (.exe and .dll files), and finds all types that extend SceneNode.
63     private static void FillTypeList()
64     {
65         nodeTypes = new List<Type>();
66         foreach (Assembly a in AppDomain.CurrentDomain.GetAssemblies())
67         {
```

```
68              try
69              {
70                  foreach (Type t in a.GetTypes())
71                  {
72                      if (t == typeof(SceneNode) || t.IsSubclassOf(typeof(SceneNode)))
73                          nodeTypes.Add(t);
74                      }
75                  }
76              catch { }
77          }
78      }
79  }
```

C.2 The LevelGenerator.CreateTile function

Listing C.2: The CreateTile(int xPos, int yPos)

```
209         private void CreateTile(int xPos, int yPos)
210         {
211             LevelTile newTile;
212 
213             if (LevelTiles[xPos, yPos] == null)
214             {
215                 newTile = new LevelTile(xPos - MaxTiles, yPos - MaxTiles);
216                 LevelTiles[xPos, yPos] = newTile;
217                 CreatedTiles.Add(newTile);
218             }
219             else
220                 return;
221 
222             // Find forced edges in adjoining tiles
223             FindForcedEdges(xPos, yPos, newTile);
224 
225             // Find an appropriate tile in our list of available tiles
226             newTile.RandomizeTile(this.availableSceneNodes);
227 
228             // Which edges still don't have a defined edge (as either edge or blocked)
229             List<Rotation> availableEdges = new List<Rotation>();
230             if (newTile.TopEdge == EdgeState.None)
231                 availableEdges.Add(Rotation.Up);
232             if (newTile.RightEdge == EdgeState.None)
233                 availableEdges.Add(Rotation.Right);
234             if (newTile.BottomEdge == EdgeState.None)
235                 availableEdges.Add(Rotation.Down);
236             if (newTile.LeftEdge == EdgeState.None)
237                 availableEdges.Add(Rotation.Left);
238 
239             // We might as well bail out here in stead of making a whole lot of calculations and instantiations
                    below...
240             if (availableEdges.Count == 0)
241                 return;
242 
243             // Shuffling
244             int n = availableEdges.Count;
245             Rotation temp;
246             while (n > 1)
247             {
248                 int k = Util.Random.Next(n);
249                 n--;
250                 temp = availableEdges[n];
251                 availableEdges[n] = availableEdges[k];
252                 availableEdges[k] = temp;
253             }
254 
255             // How many edges do we need to worry about
256             int newEdges = availableEdges.Count;
257 
258             // ...of course make sure we don't make too many tiles
259             if (AverageTiles - CreatedTiles.Count < newEdges)
260                 newEdges = AverageTiles - CreatedTiles.Count;
261 
262             // Make sure newEdges stays above -1
263             newEdges = (newEdges < 0) ? 0 : newEdges;
264 
265             // If we've been capped, we need to make sure the level actually ends somewhere!
266             if (newEdges != availableEdges.Count)
267             {
268                 newTile.TopEdge = (newTile.TopEdge == EdgeState.None) ? EdgeState.Blocked : newTile.TopEdge;
269                 newTile.RightEdge = (newTile.RightEdge == EdgeState.None) ? EdgeState.Blocked : newTile.
                        RightEdge;
```

```
270                 newTile.BottomEdge = (newTile.BottomEdge == EdgeState.None) ? EdgeState.Blocked : newTile.
                        BottomEdge;
271                 newTile.LeftEdge = (newTile.LeftEdge == EdgeState.None) ? EdgeState.Blocked : newTile.LeftEdge;
272                 return;
273             }
274 
275             int bushiness = 50;
276             // Figure out what edges should be along the tile at the available sides
277             for (int i = 0; i < newEdges; i++)
278             {
279                 switch (availableEdges[i])
280                 {
281                     case Rotation.Up:
282                         if (Util.Random.Next(100) < bushiness && CreatedTiles.Count < MaxTiles)
283                         {
284                             newTile.TopEdge = EdgeState.Edge;
285                             newTile.TileAbove = LevelTiles[xPos, yPos - 1];
286                         }
287                         else
288                             newTile.TopEdge = EdgeState.Blocked;
289                         break;
290                     case Rotation.Right:
291                         if (Util.Random.Next(100) < bushiness && CreatedTiles.Count < MaxTiles)
292                         {
293                             newTile.RightEdge = EdgeState.Edge;
294                             newTile.TileRight = LevelTiles[xPos + 1, yPos];
295                         }
296                         else
297                             newTile.RightEdge = EdgeState.Blocked;
298                         break;
299                     case Rotation.Down:
300                         if (Util.Random.Next(100) < bushiness && CreatedTiles.Count < MaxTiles)
301                         {
302                             newTile.BottomEdge = EdgeState.Edge;
303                             newTile.TileBelow = LevelTiles[xPos, yPos + 1];
304                         }
305                         else
306                             newTile.BottomEdge = EdgeState.Blocked;
307                         break;
308                     case Rotation.Left:
309                     default:
310                         if (Util.Random.Next(100) < bushiness && CreatedTiles.Count < MaxTiles)
311                         {
312                             newTile.LeftEdge = EdgeState.Edge;
313                             newTile.TileLeft = LevelTiles[xPos - 1, yPos];
314                         }
315                         else
316                             newTile.LeftEdge = EdgeState.Blocked;
317                         break;
318                 }
319             }
320 
321             // Create the new adjoining tiles
322             for (int i = 0; i < newEdges; i++)
323             {
324                 switch (availableEdges[i])
325                 {
326                     case Rotation.Up:
327                         if (newTile.TopEdge == EdgeState.Edge)
328                             CreateTile(xPos, yPos - 1);
329                         break;
330                     case Rotation.Right:
331                         if (newTile.RightEdge == EdgeState.Edge)
332                             CreateTile(xPos + 1, yPos);
333                         break;
334                     case Rotation.Down:
335                         if (newTile.BottomEdge == EdgeState.Edge)
336                             CreateTile(xPos, yPos + 1);
337                         break;
338                     case Rotation.Left:
339                     default:
340                         if (newTile.LeftEdge == EdgeState.Edge)
341                             CreateTile(xPos - 1, yPos);
342                         break;
343                 }
344             }
345 
346             // Ensure the edges are fully punched through
347             FindForcedEdges(xPos, yPos, newTile);
348         }
```

C.3 The BloodGenerator class

Listing C.3: GameEngine/TextureGeneration/BloodGenerator.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using Microsoft.Xna.Framework.Graphics;
6  using Microsoft.Xna.Framework;
7
8  namespace GameEngine.TextureGeneration
9  {
10     public class BloodGenerator : ITextureGenerator
11     {
12         /// <summary>How much liquid are we throwing at the surface in ml (influences the size of the blood
               spatter)</summary>
13         private int liquidAmount;
14         public int LiquidAmount
15         {
16             get { return this.liquidAmount; }
17             set { this.liquidAmount = (int)MathHelper.Clamp(value, 20f, 8192); }
18         }
19
20         /// <summary>The angle of the surface we have splattered blood upon. 0 is floor, PiOver2 is a wall.</
               summary>
21         private float slope;
22         public float Slope
23         {
24             get { return this.slope; }
25             set { this.slope = MathHelper.Clamp(value, 0f, MathHelper.PiOver2); }
26         }
27
28         private Texture2D texture;
29         private int realTextureSize;
30         public int TextureSize { get { return this.realTextureSize; } }
31         private GraphicsDevice gd;
32         public GraphicsDevice GraphicsDevice { set { this.gd = value; } }
33
34         private float[,] bloodTiles;
35         private int canvasSize;
36
37         /// <summary>
38         /// Create a blood generator, which by default will create a circular blood spatter
39         /// </summary>
40         /// <param name="gd">The painting graphics device</param>
41         public BloodGenerator(GraphicsDevice gd)
42         {
43             this.LiquidAmount = 1024;
44             this.Slope = 0;
45             this.gd = gd;
46         }
47
48         private void PaintDot(int xPos, int yPos, double amount, int dataWidth, Color[] data)
49         {
50             float redAmount = (float)(amount * 5d);
51
52             Color redColor = new Color(1f, 0f, 0f, redAmount);
53
54             data[(xPos * dataWidth) + yPos] = redColor;
55
56             /*if (xPos == dataWidth - 1)
57                 data[(xPos * dataWidth) + yPos] = Color.Black;
58             if (yPos == dataWidth - 1)0
59                 data[(xPos * dataWidth) + yPos] = Color.Black;*/
60         }
61
62         private void Iterate()
63         {
64             float seepageDiagonal = .6f;
65             float seepageSlopeAdjustmentUp = 1f - (this.slope / MathHelper.PiOver2);
66             float seepageSlopeAdjustmentDown = 1f + seepageSlopeAdjustmentUp;
67
68             float currentSeepage, diagonalSeepage;
69             int xPos, yPos;
70             for (int j = 0; j < this.canvasSize - 2; j++)
71             {
72                 for (int i = 0; i < this.canvasSize - 2; i++)
73                 {
74                     xPos = j + 1;
75                     yPos = i + 1;
76
77                     if (bloodTiles[xPos, yPos] > 0)
78                     {
```

```
                        currentSeepage = bloodTiles[xPos, yPos] / 10f;
                        diagonalSeepage = currentSeepage * seepageDiagonal;
                        // y-1
                        {
                            // x-1
                            bloodTiles[xPos - 1, yPos - 1] += (diagonalSeepage * seepageSlopeAdjustmentUp);
                            // x
                            bloodTiles[xPos, yPos - 1] += (currentSeepage * seepageSlopeAdjustmentUp);

                            // x+1
                            bloodTiles[xPos + 1, yPos - 1] += (diagonalSeepage * seepageSlopeAdjustmentUp);

                            bloodTiles[xPos, yPos] -= 3 * (diagonalSeepage * seepageSlopeAdjustmentUp);
                        }

                        // y
                        {
                            bloodTiles[xPos - 1, yPos] += currentSeepage;
                            bloodTiles[xPos + 1, yPos] += currentSeepage;
                            bloodTiles[xPos, yPos] -= 2 * currentSeepage;
                        }


                        // y+1
                        {
                            // x-1
                            bloodTiles[xPos - 1, yPos + 1] += (diagonalSeepage * seepageSlopeAdjustmentDown);
                            // x
                            bloodTiles[xPos, yPos + 1] += (currentSeepage * seepageSlopeAdjustmentDown);
                            // x+1
                            bloodTiles[xPos + 1, yPos + 1] += (diagonalSeepage * seepageSlopeAdjustmentDown);

                            bloodTiles[xPos, yPos] -= 3 * (diagonalSeepage * seepageSlopeAdjustmentDown);
                        }
                    }
                }
            }
        }

        #region ITextureGenerator Members

        public void Generate()
        {
            canvasSize = (this.LiquidAmount / 10);
            int halfWidth = canvasSize / 2;

            int spatterCount = this.liquidAmount / 30;
            int spatterAmount = (int)((float)this.LiquidAmount / (float)spatterCount);
            int spatterRadius = (int)((float)halfWidth * 0.60f);

            realTextureSize = 2;
            do
                realTextureSize *= 2;
            while (realTextureSize < canvasSize);
            // SurfaceFormat.Vector4 = alpha, blue, green, red
            this.texture = new Texture2D(gd, realTextureSize, realTextureSize, 0, TextureUsage.None,
                 SurfaceFormat.Color);

            this.bloodTiles = new float[canvasSize, canvasSize];

            int xPos, yPos;
            double angle, currentRadius;
            for (int i = 0; i < spatterCount; i++)
            {
                angle = Util.Random.NextDouble() * (2 * MathHelper.Pi);
                currentRadius = ((double)i / (double)spatterCount) * (double)spatterRadius;

                xPos = (int)Math.Ceiling((currentRadius * Math.Cos(angle)));
                yPos = (int)Math.Ceiling((currentRadius * Math.Sin(angle)));

                // Add an appropriate amount of liquid (that is, the amount of blood in a single spatter)
                 bloodTiles[xPos + halfWidth, yPos + halfWidth] += spatterAmount;
            }


            for (int i = 0; i < spatterCount / 4; i++)
            {
                this.Iterate();
            }
        }

        public Texture2D GetTexture()
        {
            if (texture == null)
                this.Generate();
```

```
163
164             float maxAmount = 0f;
165             for (int i = 0; i < this.canvasSize; i++)
166             {
167                 for (int j = 0; j < this.canvasSize; j++)
168                 {
169                     maxAmount = (bloodTiles[i, j] > maxAmount) ? bloodTiles[i, j] : maxAmount;
170                     if (maxAmount > 5f)
171                         break;
172                 }
173                 if (maxAmount > 5f)
174                     break;
175             }
176
177             Color[] data = new Color[realTextureSize * realTextureSize];
178             for (int i = 0; i < data.Length; i++)
179                 data[i] = Color.TransparentBlack;
180
181             int offset = (realTextureSize - canvasSize) / 2;
182
183             double theDot, theValue;
184             for (int i = 0; i < this.canvasSize; i++)
185             {
186                 for (int j = 0; j < this.canvasSize; j++)
187                 {
188                     theDot = (double)MathHelper.Clamp(this.bloodTiles[i, j], 0f, 5f);
189                     theValue = theDot / (double)maxAmount;
190                     PaintDot(i + offset, j + offset, theValue, this.realTextureSize, data);
191                 }
192             }
193
194             texture.SetData(data);
195             texture.GenerateMipMaps(TextureFilter.Linear);
196             return this.texture;
197         }
198
199         #endregion
200     }
201 }
```

C.4 XmlTilesetImporter Import Function

Listing C.4: Importing Tilesets

```
1  public override TilesetWrapper Import(string filename, ContentImporterContext context)
2  {
3      XmlDocument doc = new XmlDocument();
4      doc.Load(filename);
5      NodeList nodes = XmlTest.LoadScene(doc);
6
7      TilesetWrapper wrapper = new TilesetWrapper();
8      wrapper.Tilesets = doc;
9
10     foreach (var node in nodes)
11     {
12         TilesetNode n = node as TilesetNode;
13         if (n == null)
14             continue;
15
16         if (!wrapper.Tiles.ContainsKey(n.Name))
17             wrapper.Tiles.Add(n.Name, new Dictionary<TileType, List<XmlDocument>>());
18
19         if (!wrapper.SpecialTiles.ContainsKey(n.Name))
20             wrapper.SpecialTiles.Add(n.Name, new Dictionary<SpecialTileNodeType, List<XmlDocument>>());
21
22         foreach (var file in Directory.GetFiles(Path.Combine(Path.GetDirectoryName(filename), n.Name), "*.xml",
               SearchOption.AllDirectories))
23         {
24             try {
25                 XmlDocument t = new XmlDocument();
26                 t.Load(file);
27                 context.AddDependency(file);
28                 List<SceneNode> tNodes = XmlTest.LoadTest(t);
29                 TileInfoNode tn = tNodes.FirstOrDefault(x => x is TileInfoNode) as TileInfoNode;
30
31                 if (tn == null)
32                 {
33                     context.Logger.LogWarning("", new ContentIdentity(Path.GetFullPath(file)), "Could not add
                           tile: Xml does not contain a TileInfoNode");
34                     continue;
35                 }
```

```
36
37                  if (tn.SpecialType == SpecialTileNodeType.None)
38                  {
39                      if (!wrapper.Tiles[n.Name].ContainsKey(tn.TileType))
40                          wrapper.Tiles[n.Name].Add(tn.TileType, new List<XmlDocument>());
41                      wrapper.Tiles[n.Name][tn.TileType].Add(t);
42                  }
43                  else
44                  {
45                      if (!wrapper.SpecialTiles[n.Name].ContainsKey(tn.SpecialType))
46                          wrapper.SpecialTiles[n.Name].Add(tn.SpecialType, new List<XmlDocument>());
47                      wrapper.SpecialTiles[n.Name][tn.SpecialType].Add(t);
48                  }
49              }
50              catch(Exception e)
51              {
52                  context.Logger.LogWarning("", new ContentIdentity(Path.GetFullPath(file)), "Could␣not␣add␣tile:
                        ␣{0}", e.Message);
53              }
54          }
55      }
56      return wrapper;
57  }
```