Jan Jensen Jacob Korsgaard Jørgen Ulrik Balslev Krag Bo Lind Jensen Dan Leinir Turthra Jensen

Smart Bugs

A project by Group SP202a 2009
Aalborg University, Game and Engine Development

AALBORG UNIVERSITET

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

Title:
Smart Bugs: Would You Like to Know More?

Project theme:
AI Programming and User Experience

Project period:
Spring Semester 2009,
4th February to 4th June

Project group:
sp202a

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

Supervisor:
Zeng Yifeng

Print run:
7

Pages:
90

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

Abstract:

This report describes the development of the Smart Bugs game, which is a Half-life 2 modification. The player defends an objective against giant invading bugs, which try to learn to outsmart him, by avoiding the areas where he is most dangerous. The project focus is on working with game AI and User Experience in games.

The implemented AI is a system for representing danger on the pathfinding graph over time. Four different learning algorithms was tested, and the most suitable algorithm for the game was identified. The game was evaluated using a Playtest, which resulted in a number of improvement proposals for further development.

Even with a final algorithm for representing danger, the proper implementation into the game requires substantial tweaking to achieve a compelling experience for the player.

This report and the computer game named Smart Bugs , was developed by project group sp202a at the Aalborg University, Department of Computer Science, in the spring of 2009.

When external sources are used, the reference for the source is written in square brackets e.g. [1], where 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.

When citations are used it is explicitly noted as a citation.

A CD-ROM is attached to this report which 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 "*AI Programming and User Experience*", we are presented with an very interesting interdisciplinary task. We will be combining the technical area of artificial intelligence with the more human-orientated field of User Experience.

A requirement for this project is that the Source game engine is used for the development. In contrast to earlier semesters, this gives us the opportunity to focus on creating a game, instead of developing a game engine. However this also means, that we will be spending a substantial part of our time on learning how to use the Source engine.

Based on the requirements of the semester and the capabilities of the Source engine, we have crafted a preliminary game pitch, which briefly outlines our game idea:

> *The setting of Smart Bugs is a desert environment with mountainous rock formations and hills. The game is inspired by the movie Starship Troopers seen in figure 1.1. The player takes on the role of a lone soldier, who must defend an objective area against swarms of giant bugs.*
>
> *Smart Bugs is a First-person Shooter. The bugs will adapt to the actions of the player and actively try to avoid getting shot. As a result of this, Smart Bugs will put demands on the tactical skills of the player.*
>
> *To help the player, he is equipped with the latest in weapons technology. This includes a high powered assault rifle and light weight movable defense turrets.*
>
> *The Smart Bugs game is a Half-Life 2 modification, developed using the Source game engine.*

Figure 1.1.: Two screen captures from the movie Starship Troopers. The left is from the first assault on the bug planet. The right is from the second assault, where the bugs are attacking a human outpost on the bug planet.

1.1. Problem Statement

In order to focus the project and limit the extent, we will present an overall problem statement. Due to the interdisciplinary nature of this project, we will divide it into two questions:

1. *How can a bug learn to avoid dangerous areas in a map?*
   This question gives us the opportunity to work with topics such as intelligent agents, pathfinding, decision making and learning.

2. *How to investigate the impact such a bug has on User Experience?*
   To answer this, we will need to explore areas like User Experience, game design guidelines and game evaluation.

The two questions concern quite different academic areas, but are both relevant for the development of Smart Bugs . Solving this task will hopefully give us valuable knowledge and experience on the diverse aspects of game development.

The answer to the two questions will be presented in chapter 9 on page 80.

CHAPTER
2

Game Concept: Smart Bugs

This chapter describes the core elements and rules of the game.

Section 2.1 elaborates on the game rules, which serves as a basis for section 2.2 on the next page, which describes the overall AI in the game.

2.1. Game Elements and Rules

This section describes the concepts and rules of Smart Bugs , which will be the basis of the game design and implementation.

Bugs

Bugs are the main non-player characters (**NPCs**) in the game. A bug can deal damage to the player if it gets close enough. There are two different types of bugs in the game, which are distinguishable by their color.

Yellow bugs are the most common. They are come in large numbers and are quick learners who will change their attack paths according to the actions of the player.

Red bugs are more rare. They are faster than the yellow bugs. Red bugs have a stronger type of exoskeleton and are more resistant to fire from defense turrets. Reports indicate that the red bugs are even smarter than the yellow bugs.

Bug Waves

The bugs attack in waves. A wave a fixed amount of bugs that are spawned within a specific interval. The number of bugs vary from wave to wave.

Map Locations

A map in Smart Bugs contains two important types of locations:

A bug *spawn location* is where the bugs emerge from. They are fixed locations on the map. There are several of these in a map.

Objective locations are areas the player must defend. The bugs emerge from their spawn locations and advance toward the objective location. The objective location can only withstand a certain number of bugs reaching it before it is destroyed.

Weapons

The player has two weapons at his disposal:

The player is equipped with an *assault rifle*, which is well suited to close combat. The assault rifle has almost unlimited ammo.

A *defense turret* automatically fires at bugs within sight and range. A turret can be picked up and moved by the player. Turrets cannot be destroyed, neither by the player

nor by the bugs. However, a turret can fall over if pushed, and will require the player to stand it up again. Turrets are given to the player automatically as reinforcement at certain points in the game.

Game Rounds

The game consist of a number of rounds. At the beginning of a round the player can be granted additional turrets and a new bug wave begins. The number of bugs in the waves increase as the game progresses.

Win & Lose Conditions

- The player will *win* if both he and the objective location survives all game rounds.
- The game is *lost* if the player is killed or the objective location gets destroyed.

Game Difficulty

The game difficulty controls the hit points (**HP**) of the bugs, i.e. how much damage they can take before they die. It also affects the amount of bugs spawned during the waves.

2.2. AI Specification

The purpose of this section is to determine the overall specification for the AI in the game. This will include some overall AI design decisions and assumptions.

There are basically two requirements for the bug AI:

- Bugs must be able to find a path to the objective area.
- Bugs must be able to dynamically adjust future paths, according to the actions of the player.

To simplify the task of developing the AI, we make the following assumptions:

- **A bug can pass knowledge to other bugs when it dies**
  This scenario might not be entirely realistic, but it simplifies our task. We assume bug knowledge is globally available, so we do not need to concern ourselves with information passing between bugs.
- **The knowledge of a bug only concerns the current gaming session**
  Working with persistent data and a larger time frame for learning could be interesting, but we find it too comprehensive in proportion to what else needs to be done in this project.

Possible AI techniques for the implementation will be presented in chapter 4 on page 17.

CHAPTER
3

Source Engine Overview

This chapter is intended to give the reader an idea of what the Source game engine is and how to use it. This will be done by highlighting the features of the game engine, and the Software Development Kit (SDK), which will be used to implement Smart Bugs .

3.1. Source Architecture

Source is the FPS engine developed for Half-Life 2 by Valve. Since it was released it has been used for the development of several successful games such as Counter-Strike: Source, Portal and Left 4 Dead.

The game engine was designed primarily for FPS games and is therefore optimized for and has features facilitating such games. It has an advanced physics engine, a 3D audio engine, an NPC AI engine and network gaming support. Furthermore, the engine uses an advanced 3D rendering engine based on portal culling[1] with support for custom materials and programmable shaders. Source is a cross-platform engine which runs on Microsoft Windows, Playstation 3, Xbox, and Xbox 360. User interfaces are created using Valves VGUI2 system.

The game engine is written in C++ and can be licensed by third-party developers for their games. Alternatively, Valve has released an SDK, which is downloadable through their digital delivery service Steam. Using the SDK, developers can modify existing Valve games such as Half-Life 2 or Counter-Strike: Source, these modifications are commonly called **mods**.

It should be noted that to fully be able to create a mod that is distinct from the original Half-Life 2, multiple third party programs are needed. These are programs such as a C++ Compiler, 3D modeling application, and a sound authoring tool.

A new mod is created using the SDK tool window. This generates a directory with the SDK C++ source code which compiles into a new mod. There are two parts of creating a mod, the first and most important is creating maps using the Hammer level editor. If the mod requires new logic that cannot be created using Hammer it should be implemented in the C++ source code of the mod as new entities for Hammer to use.

The Source engine has a developer's console which can be opened from within the game. This allows developers and designers to invoke commands and change the value of some variables during runtime. These are called **Console Commands** and **Console Variables** respectively.

Besides Hammer, the SDK includes a wide range of different programs used to aid in the development of a mod. The three main tools are Hammer, Model Viewer, and Face Poser.

3.2. The Hammer Level Editor

The most important tool in the SDK is the level editor called Hammer (formerly known as WorldCraft). When creating a game using Source, a rule of thumb is that if it can be done in Hammer, do it in Hammer. Hammer is used to create the maps, or worlds, that

the player will be running around in. In Hammer geometry is used to build up the world. A basic world could be a hollow box, where the inside of the box can be covered with a skybox texture, which creates the illusion of a sky. An image of the 3D view of Hammer can be seen in figure 3.1.

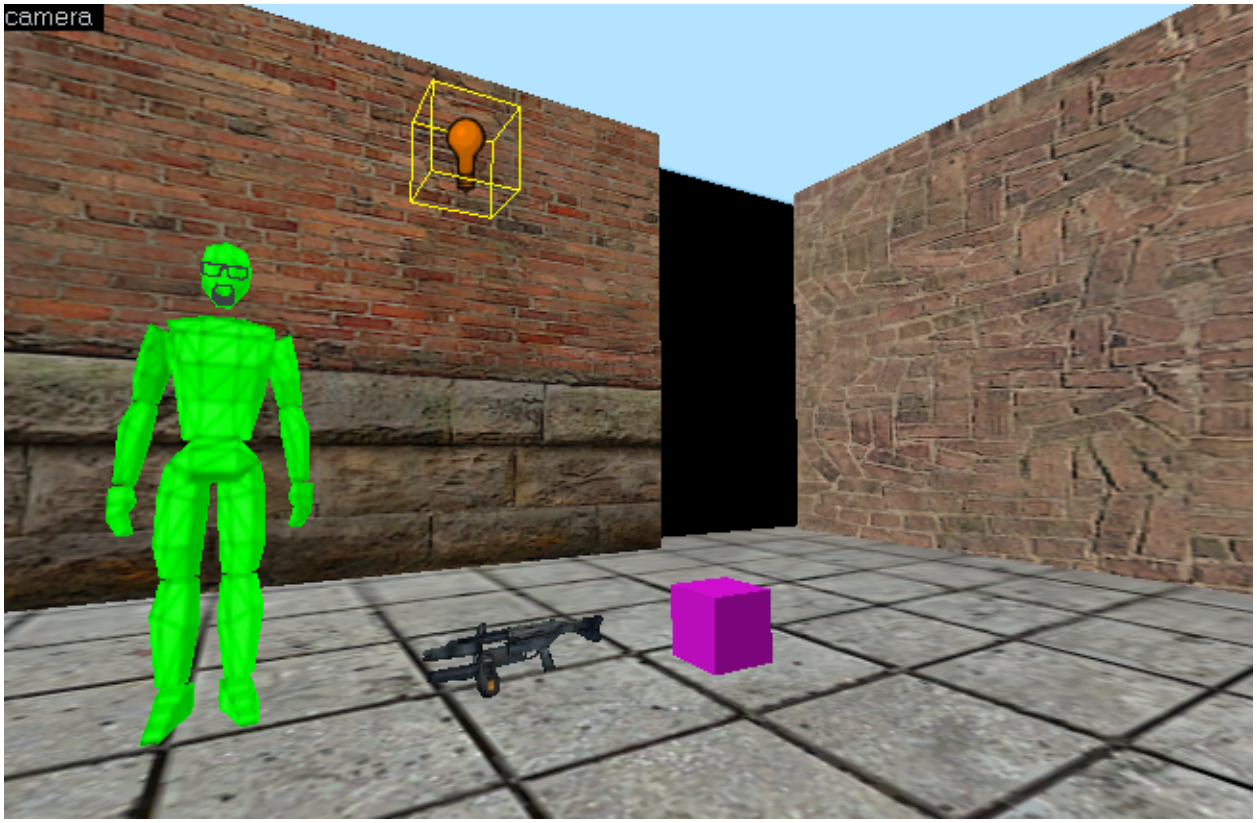


Figure 3.1.: Screenshot from hammer showing four different entities. Also in the image are four of six pieces of geometry defining the boundaries of the level, two of these are not connected leaving a hole in the wall, showing the blackness called void; this is called a leak.

Levels consist of geometry, textures, sprites, and entities (NPCs and other logical behavior). Geometry makes up the world of the game, for example a model of a house, or the ground making up the landscape. Geometry is static and will not change during the game. It can however be marked as dynamic geometry, which can be destructible, movable, etc.

Textures are applied to geometry. The same box geometry can look very different with different textures. Sprites are two-dimensional images that can be put on top of geometry; much like graffiti on a wall, sprites are not part of the geometry as textures are. Sprites add diversity to a level, and can make it more interesting.

3.2.1. Entities

An entity is a dynamic object in Source and Hammer. There are many different types of entities. Strictly logical entities do not have a visual representation within the game, but can instead provide some sort of logic for the level. More advanced dynamic objects in the game such as fully-fledged NPCs are also entities and there are many different kinds of entities with logic for features such as: AI, physics, props, environment settings, math, NPCs and Weapons.

Some commonly used entities are:

info_player_start dictates where in the level the player is to spawn when the game begins.

light is an entity which can be set to illuminate an area around it.

prop_physics is used for props that are to be influenced by physics such as tables, TV, chairs, barrels and such.

weapon_x is used to define where a given weapon should be placed in the level. X is the name of a given weapon, e.g. the weapon_pistol entity spawns a pistol.

Once placed within Hammer the entities have properties that can be customized. A light entity for example has a property defining the color of the light as seen in figure 3.2 on the following page. Entities can be set to interact with other entities, as all entities has the ability to give an Output or react to Input given from other entities.

Most entities have one or more boolean flags which can be set. These flags makes the entity behave differently according to how they are set. E.g. for the weapon_shotgun entity a flag can be set to ”Deny player pickup” which in turn will only let NPCs pick the weapon up.

Another example is the npc_combine_s representing a combine soldier. This entity has multiple flags, some of these are ”dont drop weapons”, ”ignore player push”, or ”Template NPC”. The first two flags are self explanatory. The Template NPC flag is used to tell Hammer that this NPC entity should not be spawned on level start-up, but instead be used as a template, for the npc_maker entity. This in turn is used to spawn an NPC on a specific input, sent to the npc_maker.

The functionality of an entity is implemented in C++ and exposed to Hammer using Game Data files.

3.2.2. Game Data

Hammer uses Gama Data files called Forge Game Data (FGD) files as a source of which entities can be added to a level. By default the entities of Half-Life 2 are available in a newly created Half-Life 2 mod. The FDG file defines the entities of a game, outlining their properties, help text, and appearance in Hammer.

To create a new custom entity, it is required to first program the entity logic in source code, compile the code, and create an FDG file which describes the entity to Hammer including the properties, inputs and outputs of the entity. The procedure is described in detail in section 7.2 on page 47.

3.2.3. Map Compilation

To be able to run a map created in Hammer it has to be compiled using a number of different command-line tools. Newer engines with better editors let the developer interact with a running version of the map in real time, but Hammer does not.

The Vbsp (Valve Binary Space Partition) program creates a Binary Space Partition (BSP)[1] file which contains most of the information needed by the engine to render the map. This includes all the information about the map, meaning the geometry, textures and entities.

The BSP is created by splitting up the world geometry, into *visleaves*. Leaves are split every 1024 Hammer units, meaning that geometry stretching across these limits will be

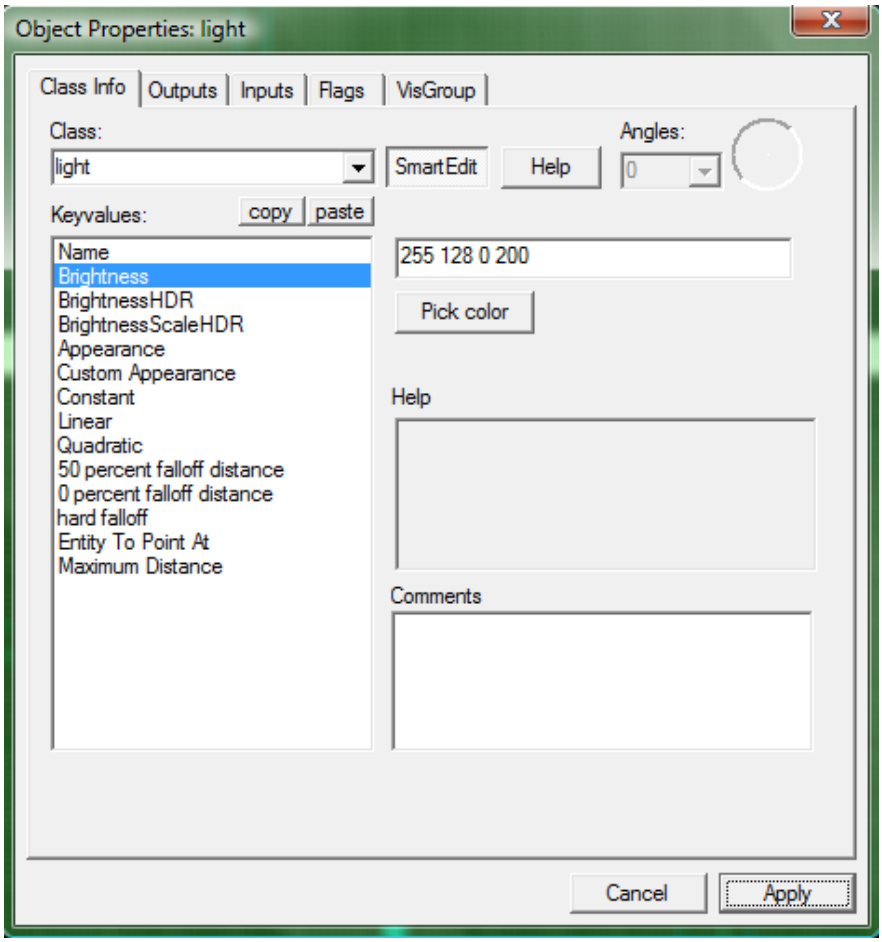


Figure 3.2.: Screenshot of a options box for the light entity. In the Class info tab the entity has been set to emit an orange color.

represented in two leaves. Displacement meshes[1] created from world geometry cannot be used for BSP calculations.

After the BSP tree has been created with Vbsp, the Vvis tool can be used to calculate visibility between the leaf nodes in the tree. Running the VIS program is optional, but highly recommended as it pre-computes visible sets (visset), meaning that it determines what the engine should render when the player is in a specific area of the map.

If Vvis is not run the engine will draw the entire map all the time instead of performing occlusion culling[1] on geometry, which is not in the current visset. So running Vvis will result in slower compile times of the map, but should end up giving faster rendering times once the map is being played. Also, if Vvis is not run there can be complications with water not being rendered properly. If the map has a leak in the geometry, as shown in Figure 3.1 on page 12, the Vvis program will not be able to tell which side of the map is inside, and which is outside, therefore no VIS calculations can be performed. If the Vvis program is not run, then the Vrad program will not know how to calculate lighting the map, and thus only basic illumination will be applied.

3.3. Model Viewer

Another important tool for content is the model viewer which lets the content creator examine the models once they have been compiled. Sometimes there are inconsistencies between what a model looks like in a 3D modeling application and what it looks like when it has been compiled for Source. The Model Viewer is used for checking that the model has textures, animation cycles and blending between animation cycles. The Model Viewer can be seen in figure 3.3.

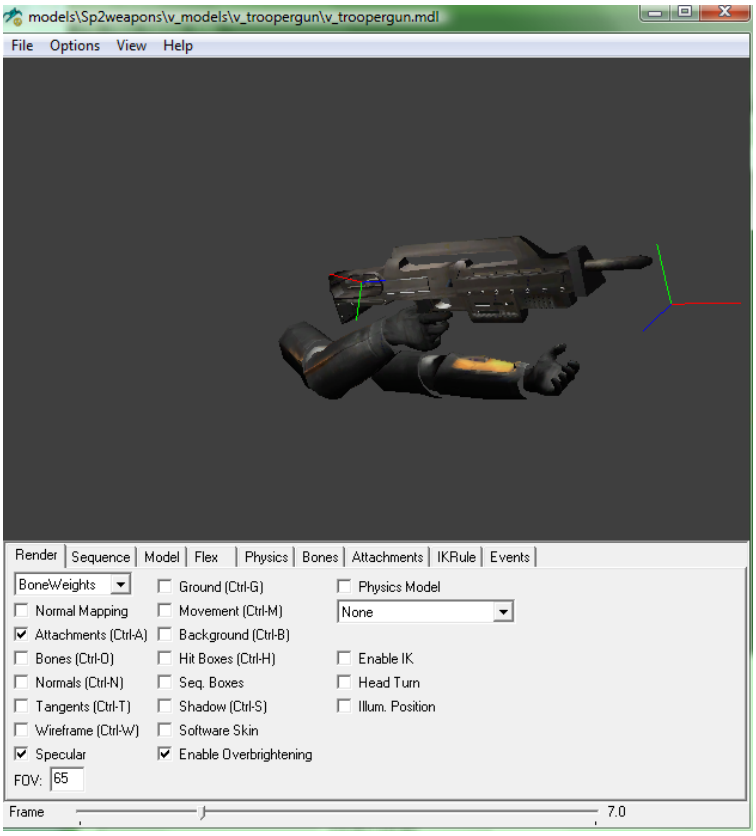


Figure 3.3.: The model viewer with a model loaded.

3.3.1. Importing Models

When importing 3D models from a third party 3D modeling application, such as Autodesk 3D Studio Max, Maya, Blender or SoftImage|XSI the models need to be exported as a StudioMdl Data (SMD) file. SMD exporters for each of the 3D applications can be found in the Source SDK.

There are three types of SMD files that needs to be exported. One is a reference file, telling Source how the model is enveloped in its initial state. The second is a physics file, used by Source to calculate how the object should collide with other objects. The last type is the animation file. Each animation should have its own animation file. Any given model can have multiple animations, but should always have at least one animation loop. The SMD files then need to be compiled into game readable model files.

To compile the SMD a .qc file is needed. The .qc file is an text file containing information on how to handle each of the animations, any attachments to the model, and the location of the texture for the model once it has been compiled. The .qc file is compiled

by the program *studiomdl*, which is also included in the SDK.

When the model has been compiled, the texture files need to be converted to a ValveTextureFormat (VTF) files. The accompanying tool for this is called vtex and converts TGA files to VTF files.

3.4. Face Poser

The last tool is the Face Poser which is used to create facial animations on existing models. The Face Poser tool makes it possible to do lip-syncing for characters, so they seem to move their lips correctly when talking in scripted events. Scripted events are created using Hammer.

CHAPTER
4

AI Techniques

In this chapter we outline which AI techniques are necessary to implement the game concept described in Chapter 2 on page 9. For each technique the general theory will be explained and there will be an overview over the existing implementation of the technique in the Source engine, if such an implementation exists.

To implement the game concept three areas of game AI will have to be considered: Pathfinding, Decision Making and Learning. Pathfinding will be needed to allow the bugs to navigate from their spawn point to the objective area, Decision Making will be needed to control the bugs and make them react to changes in their environment and Learning will be needed to make the AI learn which attack routes are to be avoided. The three areas will be discussed in the following sections.

4.1. Pathfinding

Pathfinding is an important part of AI in games. The basic idea is to represent possible navigation routes in the game world as a graph and then use search algorithms to find a path from A to B in this graph. The A* algorithm is the most commonly used pathfinding algorithm in games[8], so we focus on this.

A* Algorithm

The A* algorithm solves the problem of finding a least-expensive path between two nodes (called start and goal), in a directed non-negative weighted graph. The algorithm is guaranteed to find a path if it exists. It works by searching the graph in a best-first manner using a heuristic to estimate which node is the best to examine next.

An example of A* can be seen in listing 4.1.

Listing 4.1: A* algorithm in pseudo-code

```
Functions used:

g(x): Actual distance from the start node to node x
h(x): Estimated heuristic distance from node x to the goal node
f(x): The estimated distance from start to goal through node x
Cost(x, y): The cost of moving from x to y
Heuristic(x): The heuristically estimated distance to node_goal from node x
Solution(x): The path found by backtracking from node x until start_node is reached


The algorithm:

Initialize OPEN list to an empty list
Initialize CLOSED list to an empty list
Create goal node; call it node_goal
Create start node; call it node_start
Set g(node_start) to 0;
Set h(node_start) and f(node_start) to Heuristic(node_start)

Add node_start to the OPEN list

while the OPEN list is not empty
```

```
{
    Get the node n with the lowest f(n) from the OPEN list
    Add n to the CLOSED list
    if n is the same as node_goal we have found the solution; return Solution(n)
    Generate each successor node n' of n

    for each successor node n' of n
    {
        Set the parent of n' to n
        Set h(n') to be Heuristic(n')
        Set g(n') to be g(n) plus Cost(n, n')
        Set f(n') to be g(n') plus h(n')
        if n' is on the OPEN list and the existing one is as good or better then discard n' and
            continue
        if n' is on the CLOSED list and the existing one is as good or better then discard n' and
             continue
        Remove occurrences of n' from OPEN and CLOSED
        Add n' to the OPEN list
    }
}
```

A* Heuristics

Heuristics or heuristic methods are methods that are used to rapidly find a solution that is close to the best possible answer to a given problem. They can be perceived as rules of thumb, educated guesses or simple common sense. The heuristic method used in A* should be a function which can give a quick estimate of the cost of movement between two nodes. It also has to be **admissible** which means that it must never over-estimate the cost of reaching the goal. The most commonly used method is to use **Euclidean** distance, however to avoid the use of expensive square roots, **Manhattan** distance can be used instead. It is however not admissible without scaling all found distances. For example in 2D distances would have to scaled by $\frac{\sqrt{2}}{2}$ or by 0.7 to get an admissible heuristic[4].

4.1.1. Pathfinding in Source

In the Source engine, the NPCs use a graph called **Nodegraph** for pathfinding. NPCs navigate the Nodegraph using an A* algorithm with a Euclidian heuristic function.

Nodegraphs are created by the level designer using Hammer, by manually placing info_node entities which will become nodes in the Nodegraph. An image of a this can be seen on the left in figure 4.1 on the next page.

The Nodegraph itself is created at runtime when the map is loaded; Links are then created from each node to one another. The NPC is then able to navigate from any node to any other node using links (edges in the Nodegraph). Links can be seen on the right in figure 4.1 on the facing page.

When an NPC needs to move, it builds a route using the Nodegraph. First it finds the nearest node to its current position. Then it finds the closest node to its desired location. Finally, it finds the best path between them using A*.

4.2. Decision Making

For game NPCs to appear intelligent they need to be able to make decisions. For example, if an NPC is attacked it should be able to decide whether to retaliate or flee.

The core idea of decision making can be defined as a function which given an agents current knowledge will output an action the agent should take. There are two parts of

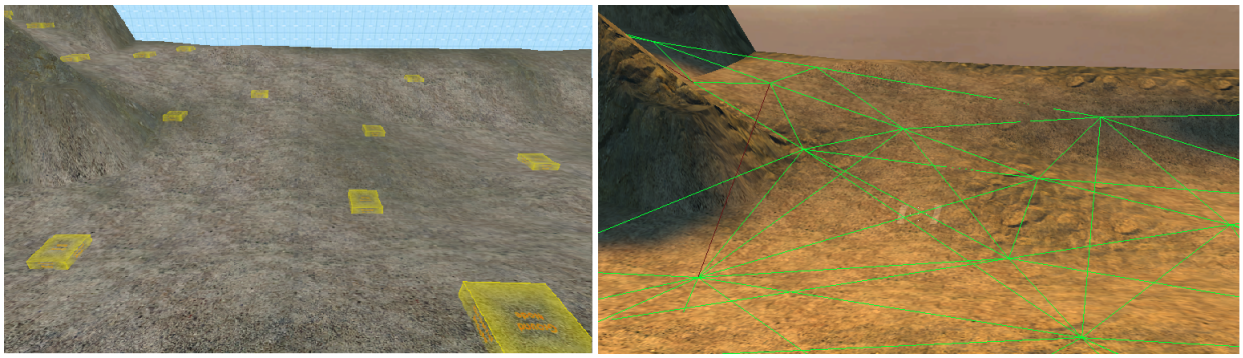


Figure 4.1.: Image showing info_nodes in hammer on the left, and a running implementation where the info_nodes are compiled into a Nodegraph, that the NPCs can use for pathfinding.

an agent's knowledge: **Internal Knowledge** represents the agent's own state and could contain its health or memory of previous events; **External Knowledge** is what the agent knows about its environment, such as the location and amount of enemies close to it. An action can be a task that the agent should perform, such as flipping a switch or attacking an enemy. It can also be an internal action which changes the agent's internal state. The amount and types of actions and knowledge is based on the requirements of the game.

An important aspect of designing a decision making system for the AI of a game is to allow level and game designers to create and modify the behavior of an agent without the involvement of a programmer. This means facilitating their understanding of the decision making process by choosing a model that can be easily visualized, is easy to understand and can be built without resorting to programming language. This frees up the programmers' (limited) time and makes the designers more productive, happier and should ultimately result in a better game.

This section outlines a few common decision making techniques before describing the specific implementation chosen by the developers of the Source engine. This section focuses solely on reactive decision making. For planning and goal oriented behavior algorithms refer to Ian Millington's book[8].

4.2.1. Finite State Machines

A simple and very common way of representing an agents internal state and actions is to use a Finite State Machine (FSM). The different behaviors or states of an agent is represented using the states in a state machine. The states are connected using transitions, each of which is associated with a condition. If a state has a transition going to another state, it will switch to the other state if the condition on the transition is met. For example, figure 4.2 on the next page shows an agent that will wander around until it sees an enemy. It will then go to the attack state until it cannot see the enemy any longer, killed the enemy or it died.

FSMs are easy to understand and powerful. Unfortunately they can become unwieldy when they become bigger. This is especially the case with Hard-Coded FSMs, which are state machines that have been implemented completely in code. A common implementation is to have an enumeration of the different states the agent can occupy and have a single function to choose which state to switch to:

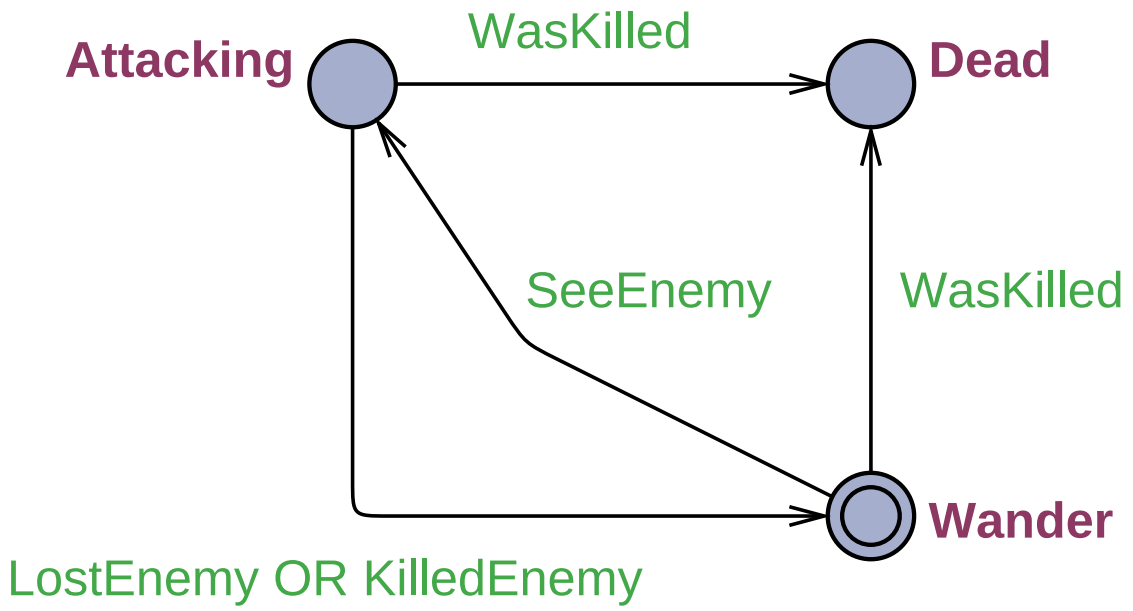


Figure 4.2.: A finite state machine of a simple agent.

Listing 4.2: A hard-coded FSM of the previous agent

```
enum NPCState
{
    Wander,
    Attacking,
    Dead
}

NPCState SelectState(NPCState currentState)
{
    if(currentState == NPCState.Dead || WasKilled)
        return NPCState.Dead;

    if(currentState == NPCState.Wander && SeeEnemy)
        return NPCState.Attacking;

    if(currentState == NPCState.Attacking)
        if(LostEnemy || KilledEnemy)
            return NPCState.Wander;
}
```

This approach will become increasingly unwieldy and difficult to maintain with the addition of more states and transitions.

4.2.1.1. Hierarchical Finite State Machines

Finite state machines do not have a construct for interrupting a behavior to do something else and then be able to return to its previous state. For example, imagine our previous agent having the ability to go to the bathroom in the middle of a fight or when wandering around. When returning from the bathroom we want the agent to be able to resume where it left off. To do this we would need a transition going from *Attacking* to a bathroom state and back when it has completed, and a similar construct from the *Wander* state as shown on figure 4.3 on the facing page. To facilitate this, a hierarchical approach can be used. The machine on figure 4.2 can be encapsulated into a behavior, which then becomes a state in an outer FSM. In the outer FSM we can put our Bathroom state, as the WasKilled transition applies to both Wander and Attacking, the result is shown on

figure 4.4 on the following page. This is called hierarchical because we can take this entire FSM and use as a single state in another FSM in which we put the ability to get killed allowing the agent to be interrupted in the Bathroom by getting murdered.

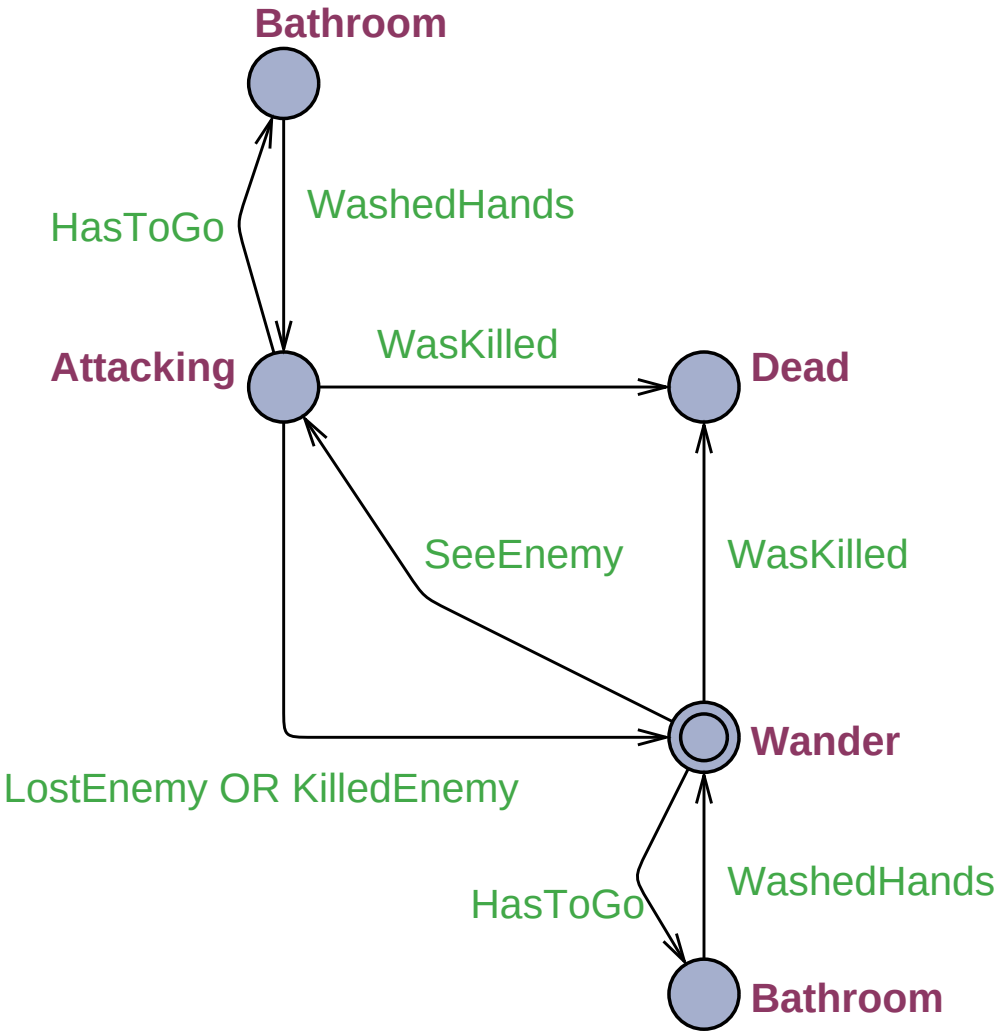


Figure 4.3.: An FSM with interrupts to go to the bathroom.

4.2.2. Scripting

Another technique used in decision making is scripting. With scripting each agent is associated with a script, which is a piece of high-level code. This technique has been quite popular in the past. It allows the developers to extend the built-in behavior of agents with specific code for specific situations or replace the built-in behavior completely without changing the code base of the game. This separation of content and code is important to facilitate large game projects as mentioned in the beginning of this section. The high-level language is often simpler and easier to work with than the language with which the game was developed, but also often carries an overhead which may be unacceptable for core functionality of the game engine.

With possibly hundreds or thousands of scripts running every frame of the game, the speed in which these scripts can be executed is essential. While scripts can be interpreted at runtime, more often they are compiled to native or byte-code for speed. The

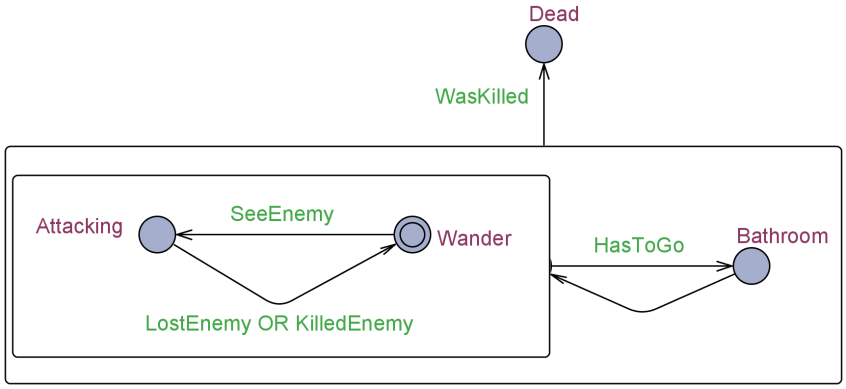


Figure 4.4.: A finite state machine of a simple agent.

byte-code can then be executed by a virtual machine, which could also have features possibly missing from the programming language the engine uses, such as garbage collection and dynamic typing. Fortunately, there are many open source languages available such as Python, Ruby and Lua, which are being used in the industry for scripting and therefore interpreters and compilers are readily available.

The scripts are given access to an API to the game engine which allows them to interact with elements of the game. Often this involves access to the scene graph as well as the physics and rendering system of the game.

A great advantage of scripting is the possibility for independent developers to extend the existing game with their own agents with completely new logic.

A disadvantage is that content creators for the game will have to familiarize themselves with the scripting programming language.

4.2.3. Decision Trees

Decision trees are a representation of the decision making process formed as the common tree data structure. In a decision tree, all leaf nodes are actions to be taken, and all other nodes in the tree have conditions that select subtrees, see figure 4.5. Every time the agent has to make a decision it starts at the root node and traverses down the tree checking conditions until it reaches a leaf node in which case the action associated with the leaf node is returned.

Decision trees have the advantage of being easily visualized. In practice, a directed acyclic graph can be used instead of a tree, in which case the end nodes are actions and every other node in the graph is a condition that selects an edge to another node.

4.2.4. Decision Making in Source

All NPCs in Source use the built-in decision making system which is a hierarchical hard-coded FSM. The main concepts of the system are:

Condition A condition is a flag which defines that some condition is met. An example is COND_SEE_ENEMY which denotes that an NPC can see one of its enemies.

Task A task in Source is analogous with Actions as described previously.

State The top layer of the decision making system is **NPC States**. It is a hard-coded FSM and the interesting states are: *Idle*, *Alert*, *Combat*, *Script* or *Dead*.

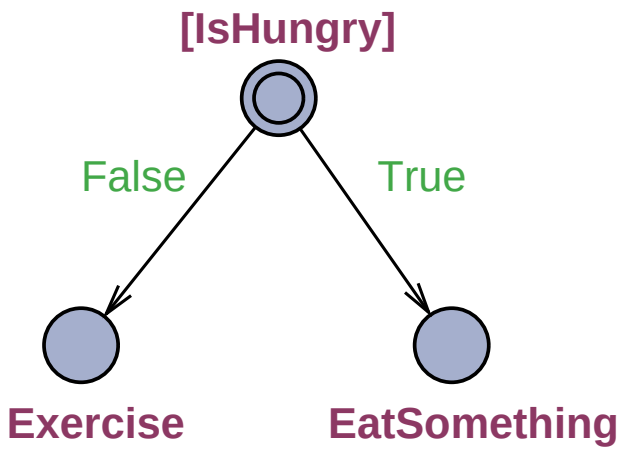


Figure 4.5.: A decision tree, the root node checks the condition IsHungry and a branch is chosen based on the result. The leaf nodes are actions to take.

Schedules Each State has associated with it a *SelectSchedule* function, which is also a hard-coded FSM which selects between an NPC's **Schedules**. A schedule defines a list of tasks the NPC should perform, and a list of conditions which if met will interrupt the schedule.

An NPC's decision making process starts when the engine calls the NPC's *Think* function. This will prompt the NPC to gather information about it surroundings using *Sensing*. Each NPC has a visual, an auditory and an olfactory system. When these systems have been updated a *GatherConditions* method is called in which condition flags are set or unset according to the NPC's internal and external knowledge. If the NPC is currently running a *Schedule* it will continue to do so unless one of the interrupt conditions are met in which case the NPC will verify its current *NPC State*, possibly changing to a new state. Then *SelectSchedule* is called.

Each NPC can be in one NPC State at a time, the states are defined in an enumeration. When an NPC is spawned into the world it starts out in the *Idle* state, if alerted to a potential threat in its vicinity it will switch to the *Alert* state. Once in the *Alert* state the NPC will not go back to *Idle* unless specifically programmed to as part of a **Schedule** or a script. If the NPC spots an enemy it will switch to the *Combat* state. Mapmakers can override the behavior of an NPC using a script, putting the NPCs into the *Script* state, the script decides when the NPC will exit this state again, and which state to go to in that case. Finally, when an NPC is killed it goes to the *Dead* state.

Once a schedule has been selected and is running, the NPC will start from the first task in the list, perform it and continue to the next until there are no more tasks or the schedule was interrupted. To start a task *StartTask* is invoked, if the task takes only one frame to complete this call will end with a *TaskComplete* call. If however the task takes several frames, the *RunTask* method on the NPC will be called each frame until *TaskComplete* is invoked as a part of the task.

4.3. Learning

Learning in computer games is used to make the game provide an extra challenge. This can be done by having the game learn the player's favorite tactics and tricks and deal with them. It can also be used to create more believable characters, who can learn from

their surrounding environment and use it to their best advantage.

Learning can be done either **online** or **offline**. Online learning means that the learning takes place while the game is running and characters using this type of learning can adapt their behavior dynamically to how the player plays the game. Offline learning is performed when the game is not running, usually between levels or before the game is released.

One of the problems with online learning is that it makes a game hard to test. If the behavior of a character depends on what the player has done, an error can be very hard to duplicate without performing the exact same sequence of actions again. Therefore offline learning is more often used as it can provide the game with more static behavior models which can be tested.

Another problem with learning algorithms is that they are hard to get right and it often requires a lot of tuning to get them to perform as intended[8]. It is often found that the work put into developing and fine tuning a learning system could have been spent better on creating a non-learning system which provided the player with the same response.

In Smart Bugs the bugs will need to know which regions of the map are the most dangerous. For example, this can be done by using action prediction to predict which areas are most likely to be defended by the player.

4.3.1. Action Prediction

When using learning for action prediction the game will try to anticipate what the player is going to do. This could be predicting which weapon the player is going to use or which route he will attack from. People are very bad at behaving completely randomly. Even if they specifically try to be random there will always be some kind of pattern to their behavior. It is this non-randomness action prediction AI makes use of to put up a good challenge for the player.

There are different ways of doing action prediction, the most simple of these is using raw probability. Raw probability operates under the assumption that the player will be most likely to keep doing what he is already doing. Consider a game where a player has to attack a location protected by the game AI. The player can choose one of three routes to attack from. When using raw probability the AI will simply keep a tally of how many times each of the attack routes have been used. The probabilities for each of the attack routes being used can be calculated as a probability distribution:

$$P(AttackRoute) = (\frac{n_1}{s}, \frac{n_2}{s}, \frac{n_3}{s})$$

where s is the total number of attacks, n_1 is the number of times the first route has been used, n_2 is the number of times the second route has been used and n_3 is the number of times route three has been used. It should be noted that for this system to work it would have to be initialized with a value such that $s \neq 0$, for example by setting n_1, n_2 and n_3 to 1 and s to 3, in order to avoid division by zero. The AI can then use the probabilities in this system to decide how to place it defenses, for example by sending defenders to the route with the highest probability of an incoming attack.

If a new attack occurs at route one the distribution is updated as follows:

$$s := s + 1$$
$$n_1 := n_1 + 1$$

leading to a new distribution where:

$$P(AttackRoute) = (\frac{n_1+1}{s+1}, \frac{n_2}{s+1}, \frac{n_3}{s+1})$$

This increases the probability that the next attack will occur from route one and the AI can then act accordingly.

Raw probability has the advantage of being very easy to implement but has the drawback of being very predictable from the players point of view. It is easy for the player to notice if the AI always defends the place he has attacked the most and it will be easy for him to come up with a tactic to fool the AI. Another problem is that the probabilities will become resistant to change over time. If the player in the example above spend most of the game attacking through route one and two he will be able to switch to route three and attack there for a long time before encountering any resistance.

4.3.1.1. Avoiding Resistance of Change

The above way of updating probabilities is similar to fractional updating which is an adaptation method used in Bayesian networks. The theory of fractional updating introduces the problem of resistance to change. The problem of resistance of change is that as the sample size increases each update will result in less change to the distribution. In the above example the number of attacks, s, would be the sample size. The example with the player attacking from route one and two in the start of the game and then having free reign to attack at route three later on in the game is an example of exploitation of resistance of change: The player can attack many times down route three before the probabilities is updated enough to make the AI move its defenses there.

In order to avoid resistance of change the system has to somehow forget about the oldest updates. In order to do this the sample size must be limited somehow and not allowed to grow forever. One way to do this is by fading. Fading introduces a fading factor, $q \in [0, 1]$, which is used when updating the distribution as follows:

$$\begin{aligned} s &:= sq + 1 \\ n_1 &:= n_1 q + 1 \\ n_2 &:= n_2 q \\ n_3 &:= n_3 q \end{aligned}$$

It should be noted that in order for this to work the numbers used can no longer be stored as integer, but will have to be stored used some kind of real number implementation. When using this approach the influence from the past will fade away exponentially. When $s \to \infty$ we get a sample size s^*:

$$s^* = \frac{1}{(1-q)}$$

s^* is called the effective sample size. Instead of declaring a fading factor an effective sample size can be declared instead. The corresponding optimal fading factor q^* can then be found as follows:

$$q^* = \frac{(s^*-1)}{s^*}$$

This might be more useful in games as it makes it possible to define how far back the AI's memory should last. In our example above we could chose a sample size of 10 which would then mean that the AI would roughly remember where the last 10 attacks occurred. More detailed description of adaptation and fractional updating can be found in [5].

It should be noted that the problem of resistance to change mainly occurs when raw prediction is used for online learning. In an offline scenario it can often be used successfully to for example predict which route a new player will choose to attack from.

4.3.2. Other Action Prediction Methods

There exists other methods for action prediction which focus on predicting a pattern to the player's behavior. Methods such as String Matching or N-Grams[8] can be used to predict the players behavior from a series of choices for the same action. However these methods are ill suited to Smart Bugs as the player can use turrets to defend multiple places at once and the algorithms are only capable of handling a single choice each round. Another problem is that these algorithms require a large amount of learning data before they are able to predict precisely.

4.3.3. Learning in Source

The Source game engine does not provide any framework for learning AI. This means that any implementation of a learning AI system will have to be made as an add-on to the existing AI implementation in Source.

CHAPTER
5

User Experience

User Experience (UX) is a very wide area, for which a common shared definition does not exist[6]. Our goal is to be able to provide good **User Experience** in a computer game, so we will focus on UX in relation to games.

We will discuss the UX concept in games in general and relate it to other relevant concepts in section 5.1. To get a more tangible understanding of UX in practice, we will look further into existing guidelines on how to design a game. This is discussed in section 5.2 on the next page. Finally section 5.3 on page 30 looks further into how the User Experience of a game can be evaluated.

5.1. User Experience in Games

The goal is to focus on UX in the context of games. However, as it is a field with ambiguous concepts, it will also be necessary to relate to other types of applications than games.

An approach to the field of UX is to talk about user-oriented quality assessment of technology[7]. Three different aspects of technology usage is presented in figure 5.1. It ranks **Experience** alongside **Usability** and **Functionality**.

The basic idea is that these are different aspects to consider about technology. **Functionality** refers to what the technology can do. **Usability** refers to the interaction between a user and the technology. **Experience** refers to what a user can achieve with the technology and feel while using it.

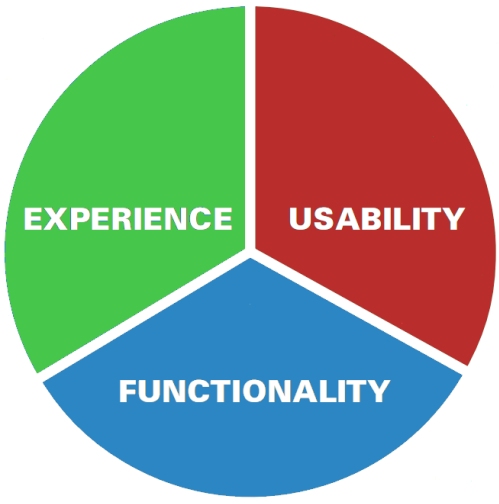


Figure 5.1.: Three aspects of technology usage[7]

A concrete example could be sending text messages with a numeric keyboard. We might agree that the Usability of such an interface is bad. However, based on the actual popularity of the technology, the experience seems to be catchy enough for the users. In this context functionality could for example be the features of the device or operator

services such as free text messages at a fixed price. The overall point of this example is that functionality, Usability and experience all are relevant, in order to assess a concrete technology. Generally speaking, assessing functionality could for example be looking at the provided features and their performance and reliability.

Assessing Usability is about measuring the ease of use of some product to accomplish tasks[9]. There exists an ISO standard which describes the concept in more detail. An aspect of the Usability concept is *Learnability*, which according to ISO 9126 is "*the capability of a software product to enable the user to learn how to use it*"[11]. In the context of a computer game, these concepts might for example refer to a user learning the interface and controls of a game.

Experience is a relevant concept in terms of games. There exist some related concepts, which may aid in understanding the field. *Playability* deals with the duration of time a game can be played[12]. This refers to the experience provided by the game, in the meaning of how entertaining the game is in the long run. This is also concerned with *Replayability*, which describes the entertainment value of playing a game more than once[13]. All these factors are also related to *Likeability* or *The Likeability Factor*, which is a more general concept concerned with whether something is compelling to a user.

Experience is also about *Sociability*, as gaming may be a social activity. This could for example be in terms of multi-player gaming or gaming communities.

The wide range of all these broad concepts may seem confusing from an overall perspective. To get a more concrete understanding of the concepts, the next section will discuss them compared to game design guidelines from literature.

5.2. Design Guidelines

The goal of this section is to discuss how the concept of UX work in practice. According to literature[3] there exist a number of guidelines on how to create a computer game. We will discuss some of these guidelines and attempt to relate them to UX and related concepts, as outlined in section 5.1 on the preceding page.

The guidelines in table 5.1 is taken from the literature review presented in [3].

Table 5.1.: Design Guidelines from Literature[3]

No.	Game Aspect	Design Guideline
#1	Game Interface	Controls should be customizable and default to industry standard settings
#2	Game Interface	The interface should be as non-intrusive as possible
#3	Game Interface	A player should always be able to identify their score/status in the game
#4	Game Interface	Follow the trends set by the gaming community to shorten the learning curve
#5	Game Interface	Interfaces should be consistent in control, color, typography, and dialog design
#6	Game Interface	For PC games, consider hiding the main computer interface during game play
#7	Game Interface	Minimize the menu layers of an interface

Continued on Next Page. . .

Table 5.1 – Continued

No.	Game Aspect	Design Guideline
#8	Game Interface	Minimize control options
#9	Game Interface	Use sound to provide meaningful feedback
#10	Game Interface	Do not expect the user to read a manual
#11	Gameplay	Feedback should be given immediately to display user control
#12	Gameplay	Get the player involved quickly and easily
#13	Gameplay	There should be a clear overriding goal of the game presented early
#14	Gameplay	There should be variable difficulty levels
#15	Gameplay	There should be multiple goals on each level
#16	Gameplay	A good game should be easy to learn and hard to master
#17	Gameplay	The game should have an unexpected outcome
#18	Gameplay	Artificial intelligence should be reasonable yet unpredictable
#19	Gameplay	Game play should be balanced so that there is no definite way to win
#20	Gameplay	The game must maintain an illusion of winnability
#21	Gameplay	Play should be fair. The game should give hints, but not too many
#22	Gameplay	The game should give rewards
#23	Gameplay	Pace the game to apply pressure to, but not frustrate the player
#24	Gameplay	Provide an interesting and absorbing tutorial
#25	Gameplay	Allow players to build content
#26	Gameplay	Make the game replayable
#27	Gameplay	Create a great storyline
#28	Gameplay	There must not be any single optimal winning strategy
#29	Gameplay	Should use visual and audio effects to arouse interest

Guidelines **#1** to **#10** are closely related to Usability, as they aim to make it easier for the game user. The concept of Learnability can for example be seen in the guidelines **#1**, **#4**, **#10** and **#15**.

A lot of the guidelines may help the Playability of a game. Examples of this are guidelines **#15** and **#22**. Guideline **#26** is directly minded on game Replayablity, but also the option of variable difficulty level (**#14**) and no single winning strategy (**#28**), may encourage users to replay the game.

The Sociability aspect is not directly represented, but guideline **#25** about allowing players to build content could be relevant. For example by creating own levels and sharing them in some online community.

The more broad concept of Likeability can be seen as all the guidelines, as every guideline has the potential to improve how well a game is perceived.

It is important to notice that these are only guidelines, a concrete game may need to diverge from them, in order to succeed. This brings us to the next topic: How do we know what works in a specific game? This is the topic of the next section.

5.3. User Experience Evaluation

We know the basic theory of UX, as it was discussed in section 5.1 on page 27. Based on section 5.2 on page 28 we have practical guidelines on how to design a game. But how do we know if all this works? To deal with this question, this section discuss possible approaches for evaluating UX in games.

The evaluation of software by involving users is not a new concept. There already exist a number of techniques for such. We will explore a number of techniques and discuss their eligibility for evaluating games.

The main difference between evaluating a game and some standard software application is their intended purpose. A game is in most cases aimed for entertainment purpose, while other software may be aimed towards solving tasks.

5.3.1. Focus Groups

One way to get input to a game is by using focus groups. The basic idea is to gather a group of people and ask them their opinions and views on for example a concept or product[10].

This makes sense for game concepts and preliminary ideas for games. The advantage of the method is that is it simple and easy to apply. The disadvantage is that the feedback gained might not be specific and there is generally a long way from concept to implementation [2].

5.3.2. Retrospective Surveys

The idea behind surveys is to get input about a specific subject. In a game context, it could for example be on which types of features players like and dislike in certain types of games.

The method is good for collecting large quantities of data. For example by sending questionnaires out to thousands of households. Creating a good questionnaire is difficult and the results of the survey depends on the quality of the questionnaire.

It is applicable for games, but the main problem is that the method is retrospective[2]. People are asked about their previous experience or opinions about things, which may have varied greatly. This makes it hard to generalize the results of the survey into something useful.

5.3.3. Beta Testing

The idea behind beta testing is to test a new product under actual usage conditions, before it is officially released[14].

For games this is usually done by very experienced players, which potentially has an impact on the kind of feedback provided[2]. They might for example find obscure bugs and gameplay balance problems, but overlook problems which could affect less experienced gamers.

5.3.4. Usability Testing

The basic idea behind Usability testing is to test systems, by letting users solve tasks and monitoring their effort in the process [9]. This is often done in special environments,

where the conditions for monitoring are optimal. This often includes several video cameras and screen recording equipment.

The method is expensive in terms of required resources for planning and conducting tests and analyzing the results afterward. The results are often very qualitative, which can be a problem if testing early versions of games[2]. The reason is that detailed feedback on Usability problems might be irrelevant, if the overall game concept does not work. Therefore doing Usability tests will only make sense late in a development process.

5.3.5. Playtesting

The Playtest method is described by John P. Davis et al.[2]. The overall approach of the method is denoted as "Surveys & Play". The basic idea is to get participants to play the game and provide feedback.

Compared to a conventional Usability test, the Playtest method yields more quantitative data. The advantage of the method is that it is relatively cheap in terms of planning, execution and analyzing results. Planning consists of creating the survey and finding the participants for the test. Execution is letting the participants try out the game on their own, no larger degree of monitoring is necessary.

In general the relative low cost of Playtesting is especially relevant in for example iterative development cycles. A sample size of up to 25-35 participants is recommended for Playtesting AAA-titles[1].

A test routine can look as follows: First a number of participants are selected. They should all belong to the intended target audience of the game. They are invited to play the game and answer a survey about the played game.

The game is supposed to be played without receiving any help, except for instructions like a game manual. This makes it possible to evaluate a larger group of participants at once, compared to for example conventional Usability testing.

A Playtest can yield various types of results. It depends on the survey presented to the participants. For example can two different variations of a computer game be compared, by simply presenting half of the users with one variation, and the other half with another variation.

By comparing the results of the survey, one should be able to tell how the different game variations affects the participants. This could for example be used for evaluating whether some specific game mechanic is well-received by the players or not.

A disadvantage of the method is that the quality of the survey is very important, which is similar to the retrospective surveys described earlier. It is also important that every participant gets the same conditions and information, in order to be able to generalize about the results.

[1] A high-quality game with a high budget

CHAPTER
6

SMART BUGS Design

This chapter is split into three main sections. In the first section we explain the reason behind the layout in which the game takes place, which influences the mood of the game. This is done in the Visual Design section 6.1. Once the visuals have been determined we take a look at the underlying mechanics needed to make the game work. These include how the bug NPCs handle damage and path finding mechanics. Also the Brainbug, which is our custom implemented entity for controlling the game, is explained. This is done in the Technical Design section 6.2 on the next page. Finally in the Map Design section 6.3 on page 40 we decide the layout of the map, while the idea of rounds and which messages to use for each, are finalized.

When designing a game for the Source Engine it is important to consider how much of the game can be created using the existing code, entities and tools available from the map editor *Hammer*. The purpose of the design is to identify which new entities must be created in source code, but also to define how these fit into the map and interact during runtime. The design also encompasses the visual design of the game, what the user is presented with while playing the game. Finally, this chapter will describe the design of a map for the game. The entire design is based on the game concept detailed in chapter 2 on page 9.

6.1. Visual Design

As described in chapter 1 on page 7, the concept of Smart Bugs is based roughly around the world of the movie Starship Troopers, and should have a visual expression similar to the movie as seen on the front cover of this report. To achieve this goal, the typeface Mobile Infantry (see figure 6.1) is used for the on-screen information - health and objective health labels, and the messages shown at the beginning of a round. This typeface will be used throughout the game, following the game design guideline **#5** in table 5.1 on page 28

The screen itself is laid out in two sections, as seen in figure 6.2 on the next page.

Figure 6.1.: Mobile Infantry typeface sample

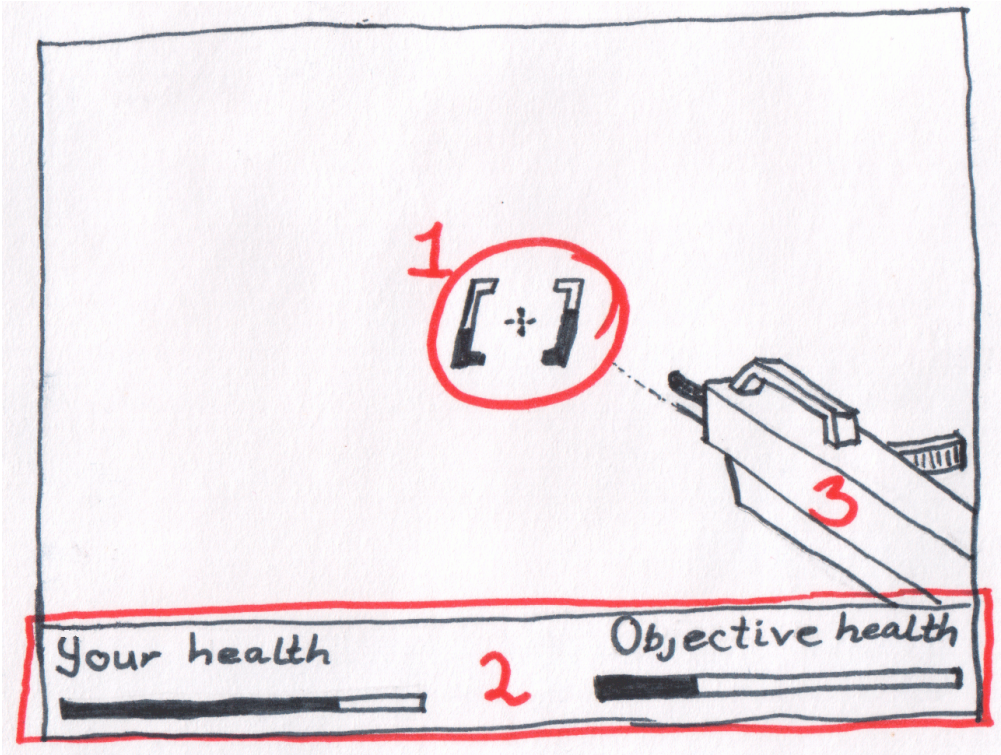


Figure 6.2.: The layout of the game screen

The top part of the screen shows the world from the player's point of view, following the game design guideline **#4** in table 5.1 on page 28. The bottom of the screen is taken up by the game information display, or HUD (Heads-Up Display).

Displayed on the HUD ② are two pieces of information: Player health and current objective health. They are shown in each their side of the display as a progress bar, which gets smaller and shows more red the less health remains.

Noticeably absent from the HUD when compared to most first-person shooters is the weapon and ammunition counts. Instead the weapon is only shown as the model shown on the screen ③ with a simple crosshair ①, following the game design guideline **#6** in table 5.1 on page 28.

6.2. Technical Design

This section describes the changes and additions which should be made to the Source SDK while creating this game. The main focus is on which entities to create, their parameters and behavior.

6.2.1. Pathfinding and Learning

The bugs in the game need to avoid areas and routes that are protected by the player and the turrets. In order to do so the bugs have to learn which areas and routes are dangerous. We can use some of the theory from action prediction to do this.

We need a way to store some information about which locations on the map are dangerous. Since we are only interested in locations which the bugs can get to it seems logical to store danger information for each of the nodes used by Source's pathfinding

system. The danger information at each node will have to be updated whenever something dangerous happens close to the node and there needs to be some system which allows danger to be forgotten or disappear over time. Making the bugs avoid dangerous locations can be done by using danger levels associated with each node in the pathfinding network to override the cost of movement used in Source's A* algorithm. This way the bugs will use the route to their goal with the least predicted amount of danger.

6.2.1.1. Registering Danger

We have decided to use the amount of damage dealt to bugs at a location as the measure for how dangerous that location is. Danger is therefore registered every time a bug takes damage. When a bug takes damage the nearest node in the pathfinding graph is found and the amount of damage dealt to the bug is added to the node's danger level.

One problem with this approach is that high-damage weapons might result in a huge amount of damage being registered for the death of a single bug, so that the death of that particular bug have the same effect as for example 10 other bugs killed with a normal-powered weapon. As this could skew up the whole danger level system the amount of damage registered will be capped to the bug's maximum amount of hit points before being put into the system.

From the AI's point of view this makes sense: It can know that a bug has died and lost the hit points it had left, but it can not know how much extra damage was dealt above the amount needed to drop the bug's hit points to zero. Whenever a bug dies we calculate the damage we want to register, r as follows:

$$r = min(a, h)$$

where a is the amount of damage dealt to the bug and h is the bug's maximum amount of hit points. When we know what amount of damage we want to register we register it at the closest node n:

$$Danger_n := Danger_n + r$$

where $Danger_n$ is the danger level at node n.

Updating a single node when a bug dies might not be enough. If the density of the nodes is high there might be nodes close by that the bugs can use and then avoid the weighted node while still taken the same general path. In order to avoid this we want to register damage not only at the nearest node but also at its connected nodes. We do this by finding all edges from the initial node and then update the nodes at the end of the edges with the initial amount of damage scaled by their distance from the initial node. We do this for each connected node, $n_{neighbor}$ by first calculating the distance, d, between the $n_{neighbor}$ and n:

$$d = dist(n, n_{neighbor})$$

where $dist$ is a distance function calculating the distance between two nodes, for example using euclidean distance. A globally defined number, m, defines the max splash damage range, which is used for two things: If a neighboring node is further away than the splash damage range no damage is registered at that node and if it is within the splash damage range m is used to calculate the registered damage at the neighboring node, $r_{neighbor}$:

$$r(1 - \tfrac{d}{m})$$

where r is the damage registered at the initial node.

6.2.1.2. Reducing Danger Over Time

As mentioned there need to be a way for registered danger to be forgotten over time. This can be done by running periodic updates on the danger level system and decrease the danger level for all the involved nodes. Since this will have a great effect on when bugs will start reusing paths which were earlier considered too dangerous we will develop different algorithms to do this so we can later test them and use the one that results in the best gameplay. Since these updates might have to be run on a huge amount of nodes it is important that they are not too complex as the update might then result in lag in the game. Below is listed four algorithms which can be used to update the danger levels.

Algorithm I This is the simplest way of removing the amount of damage stored at each location. For each update a fixed amount is simply subtracted from all locations. As a negative danger level makes no sense, locations which drop below a danger level of zero will have their value set to 0. This means that the update will be performed as follows:

$$Danger_{n,t} := max(Danger_{n,t-1} - a, 0)$$

where $Danger_{n,t}$ is the danger level at node n at time t and a is the fixed amount removed each update. With this method if we for example set a to half the amount of hit point a bug has the system will forget about the death of one bug at each location every two updates.

Algorithm II Another simple way to fade away danger is to let the damage stored at each node decay with a certain percentage each update. We introduce a decay factor $d \in [0, 1]$ and then update each node n as follows:

$$Danger_{n,t} := Danger_{n,t-1} * d$$

If we for example choose a decay factor of 0.75 this will mean the system forgets 25% of the damage dealt at each node every update.

Algorithm III With this method we want to be in control of how far back the system remembers danger at a location. For example we could say that each node should have a danger level according to the damage being dealt at that node the last 10 updates. We can do this by making a system which updates in the same way as in fractional updating as described in section 4.3 on page 23, using an effective sample size of 10. With this method we will have to store two values for each node. One value representing the value dealt at node n since the last update, $Current_n$, and another representing our learned damage at the node so far, $Memory_n$. We will also need a fading factor, q, which can be calculated from our chosen effective sample size, s^* as follows:

$$q = \frac{s^*-1}{s^*}$$

Furthermore we need a number representing the current sample size, s. The update for each node n will then be performed as follows:

$$Memory_{n,t} := \frac{Current_n + Memory_{n,t-1} * s}{(s+1)}$$
$$Current_n := 0$$

After all nodes have been updated the sample size will have to be faded according to q:

$$s := (s + 1) * q$$

For the methods mentioned earlier we could retrieve the danger level for a node simply by getting $Danger_n$ which would already contain the correct value. This will not work for this method, so $Danger_n$ will have to be calculated as follows before being retrieved:

$$Danger_n := \frac{Current_n + Memory_{n,t-1} * s}{(s+1)}$$

It should be noted that this system will not work exactly like fractional updating. This is because we are not interested in having our stored values adding up to a probability distribution. Instead we want a number representing what the danger level has been at each node over the last s^* updates, which the above will give us.

Algorithm IV With the method above as the number of updates increase, s will approach $s^* - 1$. When this happens the updating can be viewed as

$$Memory_{n,t} := \frac{Current_n + Memory_{n,t-1} * (s^* - 1)}{(s^*)}$$

Since $q = \frac{s^*-1}{s^*}$ and

$$q = \frac{s^*-1}{s^*} \Leftrightarrow s^* - 1 = qs^* \Leftrightarrow s^* - qs^* = 1 \Leftrightarrow 1 - q = \frac{1}{s^*}$$

we can rewrite this as:

$$Memory_{n,t} := Current_n * (1 - q) + Memory_{n,t-1} * q$$

This means that if we are willing to accept some trade-offs can make a simpler updating method that will work much like Algorithm III. The main difference between the two will be at the start of the game where the number of updates is low. Here this new method would react in the same way as Algorithm III if it had run for a number of updates with no damage dealt anywhere prior to the start of the game. This would mean that it would take a longer time for the bugs to learn the first dangerous routes, which might not be a bad thing from a gameplay point of view, as it would make the game easier in the beginning. Another way this could be handled would be to pre-seed the $Memory$ values for each node with values learned via offline learning from previous games.

As with Algorithm III the danger level at a node has to be calculated before it can be retrieved:

$$Danger_n := Current_n * (1 - q) + Memory_n * q$$

6.2.2. The Bug NPC

The bugs are the most important characters in the game, as they are the ones that will be seen the most and the main premise of the game is based on them being a threat. They should be implemented using the system already in place for creating NPCs in Source. This section describes the specific behavior we want the bugs to have.

The game concept calls for red bugs which take little damage from turrets and will avoid the areas where the player has dealt the most damage. This can be achieved by weighing the danger level created by the player higher than the danger created by turrets. For resistance against turret fire, a simple factor will be used. This should be based on the settings of the games current round and the difficulty level of the game.

The main behaviors of the bug NPCs are: **Burrowing**, **Advancing** and **Combat**. The FSM describing the behavior of the bug is pictured on figure 6.3 on the next page.

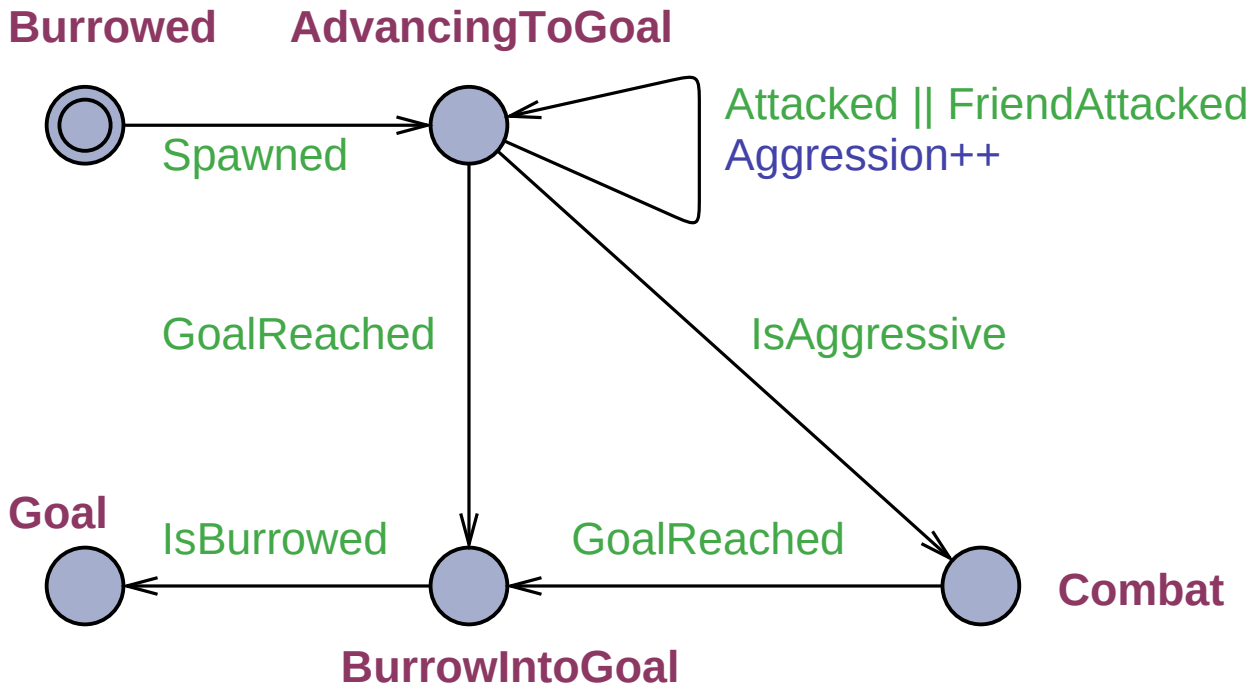


Figure 6.3.: The bug starts out burrowed, once unburrowed it will proceed toward the goal, which is the objective location. If attacked or provoked it will attack the player. If at any point the bug reaches the goal, it will attempt to burrow.

6.2.2.1. Burrowing

The bugs enter the game by burrowing out of the ground when created and when they reach the objective location they burrow into it. The bugs are spawned at map specified spawn locations, which should be places you would expect a giant insect to be able to dig out of, for example a patch of sand. The same goes for the objective location; it should also be a patch of sand where it makes sense that a bug is able to burrow into.

When unburrowing, it is important that bugs do not clump, a reasonable distance should be kept between the bugs. Otherwise it hinders local movement in Source and causes the bugs to get confused and unable to do pathfinding. The spawning mechanism should take this into account.

When burrowing into the objective location, the player should be notified by having the ground shake and the bug make a racket, according to guideline **#9** in table 5.1 on page 28. This allows quick players to sometimes kill a bug at the last moment, according to guideline **#20**.

Two *schedules* are to be implemented, one burrow and one unburrow.

6.2.2.2. Advancing

The main feature of the bug is to move from the spawning location to the objective location while avoiding dangerous areas. We are using the pathfinding graph in which each edge has a weight which is the distance between the nodes. By increasing the weights with the amount of danger at each node the built-in A* algorithm will prefer safer routes.

The bugs can either walk or run the path, this allows us to tweak the difficulty of rounds by adding more running bugs. Two *schedules* should be implemented both

using the same pathfinding algorithm: One for walking from the spawning location to the objective location, and one for running.

6.2.2.3. Combat

When attacked, the bug which was attacked should rally the nearby bugs and retaliate. When the player attacks the bug, it will remember each point of damage and use it to prioritize between advancing, and killing the player. This aggression system is very simple and widely used. The bugs will attack the enemy which has dealt it the most damage if the aggression points have exceeded a specific amount. Because turrets cannot be destroyed by the bugs, they will not accumulate aggression toward the turrets. This effectively means that the system only considers the player and how much damage he has dealt a bug.

Because bugs are easily killed by the player, they should communicate the damage dealt to them to other nearby bugs. If a few bugs are killed within a group, the group will attack the player. This is noticeable because the entire group will begin running toward the player.

To make it more visible to the player that he is being useful when killing bugs in a group that are also under fire from the turrets, the bugs should die differently when killed by the player. By simply letting player fire cause the bugs to explode into pieces, it creates a satisfying visual effect and also gives the player another incentive to actively participate in the battle, because otherwise more dead bugs will be present on the map making it more difficult to spot living ones between the carcasses.

The combat schedules are implemented in the source engine and shared among all NPCs. All that needs to be done is to mark which types of attack the bug can use and what the parameters are for these. For the bugs these are short ranged physical attacks and the range is based on the animation of the attack.

6.2.3. Brainbug and Rounds

The game concept describes waves of bugs attacking the player in a round based structure. From this requirement we define three entities: **Rounds**, **Points of Interest (PoI)** and the **Brainbug**.

6.2.3.1. Rounds

A round is a logical entity which simply contains data about a round. The Brainbug will use these entities to define the parameters of the bugs it will create. To give the level designer as much power as possible a round has a multitude of parameters:

Round Number Provides an order in the rounds, the Brainbug will execute the rounds in ascending order. If two rounds have the same round number one will be selected at random for that round.

Bugs The amount of bugs to be spawned during this wave.

Red Bugs The amount of red bugs as described in the game concept.

Turrets How many turrets to give the player in this round.

Run Probability The likelihood that a bug will run instead of walk.

Red Bug Avoidance The factor red bugs use to weigh player damage more than turret damage when calculating the danger level of a pathfinding node. The default value is 5.

Red Bug Turret Resistance The factor red bugs use to adjust turret damage. The default value of this is 0.1 meaning the red bugs take only 10% damage from turrets.

Round Delay The amount of time after all bugs have been killed before the Brainbug begins spawning bugs from the next round.

A round should also be able to notify when it is begun, using an output event.

6.2.3.2. Points of Interest

There are two different types of Points of Interest(PoI): a *start* that defines a spawn location, and an *end* which defines a objective location for the bugs to advance to. Each PoI has a radius assigned to it, which is used when spawning entities at the PoI and to determine if a bug has reached a objective location.

End PoIs can define a next end PoI, which is the next objective location the Brainbug must conquer to win the game. This allows the level designer to create multistage levels, where the player must defend a position until it is overrun and he must pull back to the next position.

Each objective location is associated with a threshold which specifies how many bugs must reach the objective before the Brainbug conquers it. Once there are no more objectives to conquer the Brainbug wins the game.

6.2.3.3. Brainbug

The Brainbug is the main game logic entity, it handles spawning of bugs setting their parameters according to the current round and assigning them an objective location to advance to. The Brainbug keeps a list of all PoIs and all rounds.

It has only one parameter which is the first objective the brainbug must conquer. To keep other entities in the map informed, the Brainbug also has these output events:

Bugs Wins Fired when the Brainbug wins. That is when there are no more objectives to conquer.

Player Wins When all bugs have been killed, the Brainbug cannot spawn any more and the threshold of the objective location has not been reached, the player has won and this event is fired.

Bug Reached Goal Fires whenever a bug reaches an objective location and manages to burrow in.

Begin Round Fires when a new round begins.

Turret Spawned Fires when a new turret is spawned for the player.

6.2.4. Turret

The turrets, as pictured on figure 6.4 on the next page, should be movable. Once placed they will attack bugs in a cone in front of them. A turret can only target one bug and will change target only when that bug is out of sight or has been killed. This makes the player's role in killing red bugs much more important as the turret will waste more time shooting the red bugs compared to when shooting the yellow bugs.

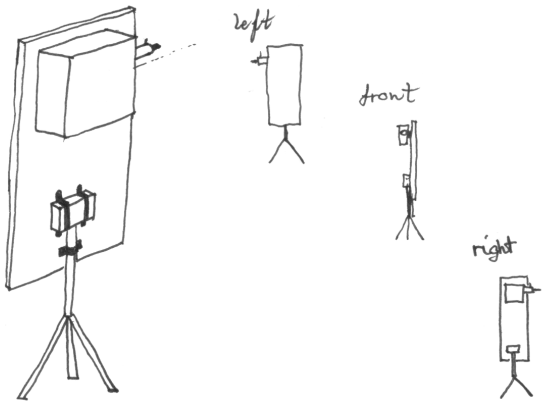


Figure 6.4.: A sketch of the turret used in Smart Bugs

6.3. Map Design

In Starship Troopers, the main battles take place on a planet by the name Klendathu, which is described as "an ugly planet; a Bug planet. A planet hostile to life as we know it." and furthermore shown to be primarily composed of rock-strewn desert landscape with many rocky lavines. For this reason, the map in Smart Bugs is designed to infuse the player a similar feeling.

6.3.1. Level Design

The basic level design, as pictured in figure 6.5 on the facing page, is that of a valley in between an outcropping of rocks, in the center of which is positioned the raised, sand-filled crater which serves as the objective for the bugs, and the position the player is to defend.

The rocky outcroppings create a series of routes, one of which a bug must follow to get from their spawn point in the mountains to the plateau on which the objective is found. They also provide shelter for the bugs as they travel around the main scene of action around the plateau, allowing them to travel at least part of the way without being under constant turret fire, which also assists the node weight calculations to adapt in a more effective manner than if each point on the map were reachable by only a couple of turrets.

The central area of the level, surrounding the raised platform, provides an arena-like area where the bugs can roam more freely after they have traveled through the paths to get around the dangerous areas, as described in section 6.2.2.2 on page 37. This is designed as the main area in which fighting with the bugs will take place for the player; While there is nothing to stop the player from going into the maze-like system of paths surrounding the plateau, he will soon be discouraged from doing this due to the amount of bugs that will then invade the target by other paths.

6.3.2. Rounds

The design of the rounds used in the game is such that the difficulty of the game grows progressively higher as the player works through them. The map contains eleven rounds.

The reasoning behind what happens in each round varies from round to round, though primarily with intensity increase in mind. The following is a list outlining the idea behind each round:

Figure 6.5.: A sketch showing the basic design of the level

1. In the first round the player is introduced to the game lightly, with a round that he cannot lose. Only yellow bugs, none of which are running. He is given one turret to defend with, and this turret is automatically placed in such a way that the first wave will have a very low rate of bugs that manage to dig in. This corresponds to game design guidelines **#10** and **#12** in table 5.1 on page 28.

2. The second round introduces the red bugs, which are introduced as being impervious to turret fire. While this is not technically true, the turrets have to deal much more damage to them than to yellow bugs. As such, the player now has to actively take part in the game. There are still not enough bugs sent out into the level for the player to lose the game, even without the turret.

3. The player is given a new turret in round three, and instructed to get it set up both by the speaker voice and by the on screen text.

4. The yellow bugs start to be able to run, increasing the pressure on the player further.

5. After the previous round, in which the player was surprised with more faster bugs, he is given a new turret to set up and help with the fight. Corresponding to game design guideline **#22**.

6. Progressive increase in difficulty, where the player is promised reinforcements by the on screen message. Corresponding to game design guideline **#23**.

7. As above. The player is granted another turret in place of actual reinforcements.

8. A jump in difficulty. The percentage of running bugs is increased less than previously, but the number of red bugs is increased greatly.

9. The number of yellow bugs is not increased at all, but the number of red bugs is increased greatly again. An additional turret is given to the player to help with the high number of bugs.

10. The amount of bugs is increased greatly, and the number of red bugs increased by the same amount as the previous two rounds. Coupled with not giving the player more turrets, this is meant to present an even greater challenge. Corresponding to game design guideline **#16**.

11. To be viewed as a type of bonus round, round 11 is designed to present the player with an even harder difficulty. He is presented with one new turret to help, but the number of bugs is increased by such a large amount, that a single turret only helps a relatively small amount. This is done to provoke the player into near-panic for the final round, creating a seemingly never-ending torrent of bugs trying to reach the target.

6.3.2.1. Messages

The list below contains the messages intended to be shown at the beginning of each round. Messages which are direct quotes from the movie are postfixed with an asterisk:

1. Don't let them burrow!

2. Your turrets are useless against red bugs - kill them yourself!

3. What she said!
4. Red ones go faster!
5. Remember - service guarantees citizenship *
6. Reinforcements are inbound!
7. Reinforcements have arrived! Here's your turret!
8. Remember your training, and you will make it out alive! *
9. I need a corporal. You're it, until you're dead or I find someone better *
10. Come on you apes, you wanna live forever? *
11. Kill 'em all! *

To alleviate potential confusion when a turret is spawned, a voice in the player's speakers will utter the phrase: "Get some turrets set up!". While this voice sample is taken from Half-Life 2: Episode 1, on which Smart Bugs is built, it is still so much in keep with the general feel of the world that it fits. This message is also the reason for the message on screen in round 3, where the text refers directly back to the spoken message. The message is furthermore delayed in this round by an amount of time large enough that the sound is done playing before the message is shown.

If the player wins, they are greeted with the quote: "It's afraid! IT'S AFRAID!" The quote is taken from the end of the movie, where a brain bug has been captured by the soldiers. If the player looses, another quote from earlier in the movie is employed, where a campaign against the bugs has failed: "- My God, how could this happen? - We thought we were smarter than the Bugs."

CHAPTER
7

Implementation

This chapter will detail the non-trivial parts of the game implementation in the Source engine.

The main principle to adhere to when programming for the Source engine is to keep everything as entities. If some functionality is needed, before modifying any existing code, verify that the functionality is not already available in an existing entity. If this is not the case then attempt to create the functionality as a new entity and only if that is impossible should changes to the existing code be made.

For Smart Bugs , most functionality is separate from the existing code base and has been implemented as entities. The following entities were implemented to be available from Hammer level editor:

sp2_bug is the bug NPC.

sp2_brainbug is the Brainbug entity, which contains the logic described in the design chapter.

sp2_round represents a round of the game.

sp2_bugpoi represents a Point of Interest as described in section 6.2.3.2 on page 39.

npc_turret_floor is the turret the player will use to help fend off bugs.

In addition to the entities already available from Hammer the danger level and pathfinding node weighting system is also implemented as entities. We will describe these entities first.

7.1. Pathfinding and Danger Levels

To modify the pathfinding system in Source with additional weights on the nodes of the pathfinding graph in a map we need to store extra data for every node in the graph. One way of achieving this is to modify the actual nodes to contain the data we need. This however goes against our main implementation principle, which is to avoid modifying existing code when possible. Instead, because nodes are represented in a list in the navigation system of Source and are associated with indices, we could simply create a list of equal size and use the node indices to access the data. This allows us to save our extra graph information in separate entities called **NodeWeight** entities.

One problem with this approach is that Source rebuilds the pathfinding graph at runtime. To allow level designers and builders to create graphs from within the game, the graph with proper indices is created after all entities in the world have been spawned, and can be rebuilt from within the game thus invalidating all indices.

To solve this problem, a NodeWeight entity is instantiated by the actual navigation system located in the class *AINetworkManager* when the game starts and a reference to the entity is saved on the manager. When the AINetworkManager rebuilds the map Nodegraph, it will reset the NodeWeight entity. This approach also solves the problem of where to place the entity for it to be accessible from the bugs that use it to navigate.

The pathfinding graph of the currently loaded map is available from the global variable *g_pAINetworkManager* and by making our NodeWeight entity a field on the AINetwork-Manager class it has been made accessible.

Each algorithm in the node weight system is implemented as a NodeWeight entity and inherits from the CNodeWeight class. The contents of the CNodeWeight class looks like this:

Listing 7.1: The CNodeWeight class

```
// Stores one vector for each node in the pathfinding network for the current map
CUtlVector<Vector> CurrentWeights;

// Initialise the internal list(s) of nodeweights
virtual void Initialize();

// Reset all weights back to the state the should have at the start of a level
virtual void Reset();

// Perform the updating of the nodewieghts system
virtual void Update() {};

// Think method used to call the update method
virtual void UpdateThink();

// Return the current weight of the node with the specified nodeIndex
virtual Vector GetNodeWeight(int nodeIndex);

// Adjust the current weight of the node with the specified nodeIndex
// by the given amount
virtual void ChangeWeight(int nodeIndex, Vector amount);

// Set the current weight of the node with the specified nodeIndex
// to amount
virtual void SetWeight(int nodeIndex, Vector amount);

// Return the number of nodes in the nodeweight system
virtual int Count();
```

A NodeWeight entity stores its weights as vectors to allow for different kinds of damage to be registered separately. This means that whenever a weight is updated or modified the change has to be done to all elements of the vector instead of just on one value. The default *GetNodeWeight* will return the value stored in *CurrentWeights* for the current node and the default *ChangeWeight* and *SetWeight* methods change this value as well.

Three different classes have been created to implement the algorithms outlined in section 6.2.1 on page 33: **CFadingNodeWeight**, **CLearningNodeWeight** and **CMergingNodeWeight**.

The CFadingNodeWeight Class

The CFadingNodeWeight class implements the methods from both **Algorithm I and II** in one class. It uses two console variables to adjust the amounts to remove and decay with: *nodeweight_fade_per_update* and *nodeweight_fade_factor* (the decay rate).

Since it only uses one value per node CFadingNodeWeight only needs to override the update method:

Listing 7.2: The Update method from the CFadingNodeWeight class

```
void CFadingNodeWeight::Update()
{
    Vector fadeVector = Vector(1, 1, 1) * nodeweight_fade_per_update.GetFloat();
    for(int i=CurrentWeights.Count()-1;i>=0;i--)
    {
        Vector newValue = CurrentWeights[i] * nodeweight_fade_factor.GetFloat();
        newValue -= fadeVector;
```

```
        if(newValue.x < 0.0)
            newValue.x = 0.0;
        if(newValue.y < 0.0)
            newValue.y = 0.0;
        if(newValue.z < 0.0)
            newValue.z = 0.0;
        CurrentWeights[i] = newValue;
    }
}
```

The class allows using any mix of fixed amount fading and decay, but can be configured to do only decay by setting *nodeweight_fade_per_update* to 0 or to only do fixed amount fading by setting *nodeweight_fade_factor* to 1.

The CLearningNodeWeight Class

The CLearningNodeWeight implements **Algorithm III**. In order to do this another list of vectors called **Buffer** is added. This implementation uses one console variable (mentioned in chapter 3 on page 11), *nodeweight_learning_sample_size*, which specifies how many updates it should consider. It is implemented as follows:

Listing 7.3: Methods of the CLearningNodeWeight class

```
void CLearningNodeWeight::Initialize()
{
    m_fNumberOfUpdates = 0.0;
    Buffer = CUtlVector<Vector>(0, g_pBigAINet->NumNodes());
    BaseClass::Initialize();
}


// Updates the weights over time
void CLearningNodeWeight::Update()
{
    float updateFactor = 1 / (m_fNumberOfUpdates + 1);
    Vector resetValue = Vector(0.0,0.0,0.0);

    for(int i=CurrentWeights.Count()-1;i>=0;i--)
    {
        Buffer[i] = (CurrentWeights[i] + Buffer[i] * m_fNumberOfUpdates) * updateFactor;
        CurrentWeights[i] = resetValue;
    }

    m_fNumberOfUpdates += 1.0;
    m_fNumberOfUpdates *= (1 - 1/nodeweight_learning_sample_size.GetFloat());

}

Vector CLearningNodeWeight::GetNodeWeight(int nodeIndex)
{
    return (CurrentWeights[nodeIndex] + Buffer[nodeIndex] * m_fNumberOfUpdates) / (
        m_fNumberOfUpdates + 1);
}

void CLearningNodeWeight::Reset()
{
    BaseClass::Reset();

    Buffer.RemoveAll();

    Vector a = Vector(0.0,0.0,0.0);
    for(int i=0;i<g_pBigAINet->NumNodes();i++)
    {
        Buffer.AddToTail(a);
    }
}
```

The CMergingNodeWeight Class

The CMergingNodeWeight implements **Algorithm IV** and works in much the same way as the CLearningNodeWeight class. It has a Buffer list and the console variable *nodeweight_merge_factor*, which is used to specify the merging factor. It is implemented as follows:

Listing 7.4: Methods of the CLearningNodeWeight class

```
void CMergingNodeWeight::Initialize()
{
    Buffer = CUtlVector<Vector>(0, g_pBigAINet->NumNodes());
    BaseClass::Initialize();
}


// Updates the weights over time
void CMergingNodeWeight::Update()
{
    float currentFactor = nodeweight_merge_factor.GetFloat();
    float bufferFactor = 1 - currentFactor;
    Vector resetValue = Vector(0.0,0.0,0.0);

    for(int i=CurrentWeights.Count()-1;i>=0;i--)
    {
        Buffer[i] = CurrentWeights[i] * currentFactor + Buffer[i] * bufferFactor;
        CurrentWeights[i] = resetValue;
    }
}

Vector CMergingNodeWeight::GetNodeWeight(int nodeIndex)
{
    return CurrentWeights[nodeIndex] * nodeweight_merge_factor.GetFloat() + Buffer[nodeIndex] *
        (1 - nodeweight_merge_factor.GetFloat());
}


void CMergingNodeWeight::Reset()
{
    BaseClass::Reset();

    Buffer.RemoveAll();

    Vector a = Vector(0.0,0.0,0.0);
    for(int i=0;i<g_pBigAINet->NumNodes();i++)
    {
        Buffer.AddToTail(a);
    }
}
```

7.2. Entities for Hammer

To make an entity available from the level builder tool Hammer, the entity class must be defined correctly in the source code. This is done by using existing C++ preprocessor macros from the Source SDK. Furthermore, the variables defined in the source should be defined in the FGD file as described in section 3.2.2 on page 13. The following is the C++ source and FGD file for an sp2_bugpoi, all entities and NPC's use this pattern to expose properties to Hammer:

Listing 7.5: Complete Source for a Bug POI

```
//////////////////////////
///// sp2_bugpoi.h ///////
//////////////////////////

#include "cbase.h"
#ifndef SP2_BUGPOI_H
#define SP2_BUGPOI_H

class BugPOI : public CLogicalEntity
{
public:
    enum POIType { Start = 0, End = 1 };

    DECLARE_CLASS( BugPOI, CLogicalEntity );
    DECLARE_DATADESC();

    // Is this a start or end POI?
    POIType m_nTypeOfPOI;

    // For end points, how many bugs need to reach it for the "objective" to be completed?
    int m_nThreshold;

    // The destination POI for this POI (the only time this is allowed to be empty is the final
        POI in a level!)
    string_t m_sDestinationPOI;

    // How far out will bugs spawn from the POI (this is used to avoid bugs just spawning on top
        of eachother)
    float m_flSpawnRadius;

    BugPOI *nextPOI;
};
#endif


//////////////////////////
///// sp2_bugpoi.cpp /////
//////////////////////////

#include "cbase.h"
#include "sp2_bugpoi.h"

// memdbgon must be the last include file in a .cpp file!!!
#include "tier0/memdbgon.h"

LINK_ENTITY_TO_CLASS( sp2_bugpoi, BugPOI );

// Expose fields to Hammer
BEGIN_DATADESC( BugPOI )
    // Links our member variables to Hammer
    // Eg.: DEFINE_KEYFIELD( localVariableName, DATA_TYPE, "fgdVariableName" ),
    DEFINE_KEYFIELD( m_nThreshold, FIELD_INTEGER, "threshold" ),
    DEFINE_KEYFIELD( m_sDestinationPOI, FIELD_STRING, "destinationPOI" ),

    // This must be 0 or 1, enums do not translate directly into FGD, but this should be POIType
        (choice)
    DEFINE_KEYFIELD( m_nTypeOfPOI, FIELD_INTEGER, "typeOfPOI" ),
    DEFINE_KEYFIELD( m_flSpawnRadius, FIELD_FLOAT, "spawnRadius" ),
END_DATADESC()


//////////////////////////
///// sp2_bugpoi.fgd /////
//////////////////////////

@PointClass base(Targetname) sphere(spawnRadius) = sp2_bugpoi : "Bug Point Of Interest"
[
    // Eg.: variableName(dataType) : "Name" : defaultValue : "Description"
    spawnRadius(float) : "Radius" : 256 : "Radius"
    destinationPOI(target_destination) : "Destination POI" : "" : "Name of the next POI. If this
```

```
        is an End point, this is the Start point it is relevant to. If this is a Start point,
        this is the next POI in the progression sequence (if a Start point has no Destination POI
        , it is the level end)."
    threshold(integer) : "Threshold" : 5 : "End type only. How many bugs need to reach this POI
        for the brainbug to progress to next POI in the progression sequence?"
    typeOfPOI(choices) : "Type of POI" : 0 : "Which type of interest point is this?" =
    [
        0 : "Start point"
        1 : "End point"
    ]
]
```

As the sp2_bugpoi and sp2_round entities have no extra functionality beyond defining data for the game, implementing the above is enough to add them. As with the built-in pathfinding nodes (info_node), these entities are trivial and superfluous without something utilizing them. The entity which makes use of these entities is the Brainbug, called sp2_brainbug.

7.3. Entity Relations and Level Progress

Figure 7.1 shows the connections between entities in a hypothetical level which contains four rounds and two progressive objectives with accompanying spawn locations.

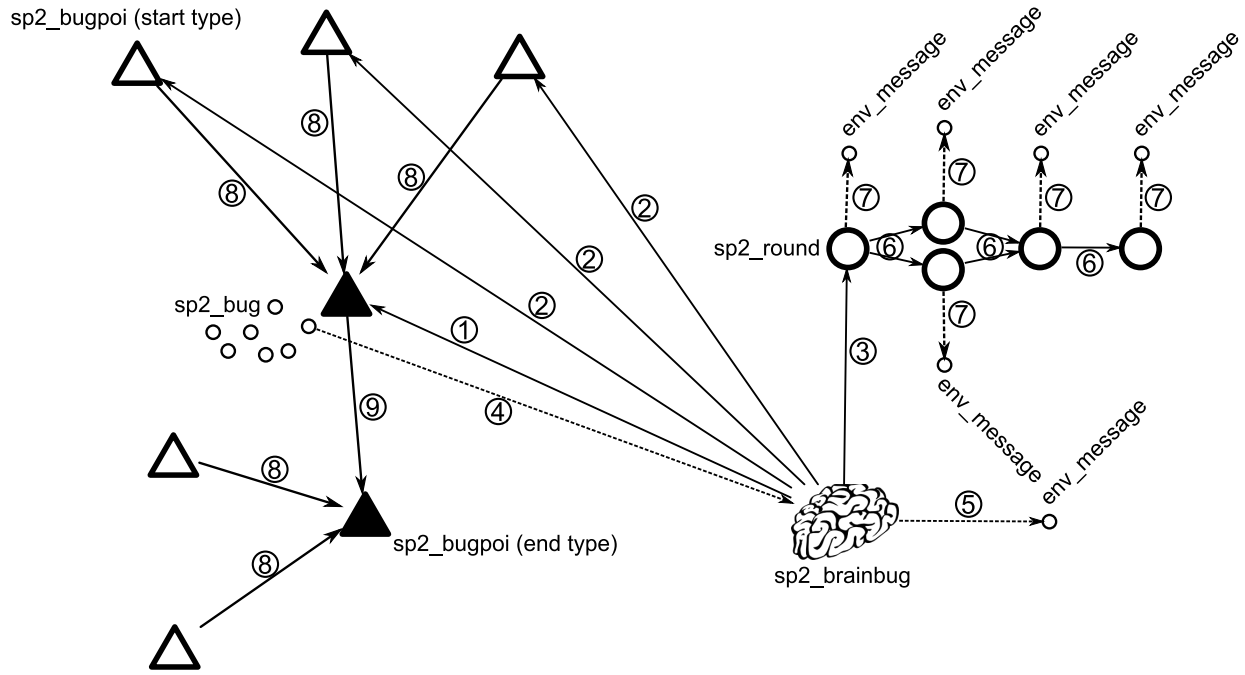


Figure 7.1.: The connections between entities in a hypothetical level. The env_message entity simply shows a message on the screen.

Upon level load, the sp2_brainbug entity gathers information on the sp2_bugpoi and sp2_round entities in the level. It points itself at the first objective location represented by an end sp2_bugpoi (①), and at the first sp2_round (③).

When the level is started, the Brainbug tells the first round that it has begun, which in turn causes the round to fire its RoundBegun output (⑦). The Brainbug then spawns the items defined by the current round parameters, first the turrets, then the bugs in random order and at a randomly selected spawn locations (in this case one of the three start sp2_bugpoi pointing (⑧) at the current end sp2_bugpoi).

The bugs advances through the level as described in section 4.1 on page 17. If they reach the objective location they dig down and inform the Brainbug that they have reached the end sp2_bugpoi by calling the ReachedPOI() function on the sp2_brainbug (④), which then counts how many times this has happened.

If the number of bugs which have reached the current end sp2_bugpoi reaches the *Threshold* property on that entity the Brainbug will progress to the next objective location in the level progression (⑨). This also unlocks the spawn locations which points to the new objective location. If the current sp2_bugpoi has no *destinationPoI* defined the Brainbug has won. In this case it first fires the *BugsWon* output (⑤) and a specified number of seconds later it restarts the level.

If the Brainbug runs out of bugs to spawn in a round, it will wait the number of seconds defined in the current round before starting the next round. To progress the Brainbug checks its cache of sp2_round entities for any rounds with the round number of the current round plus one (⑥) and adds them to a temporary list. It then chooses one of these by random. If there is only one, it will choose that one. If there are none the player has won and the output *BugsLost* is fired (⑤).

7.4. The Brainbug Entity

The Brainbug is a logical point entity, meaning it has a position in the world, but no visual representation in the game. It uses the data from the sp2_bugpoi and sp2_round. To access this data, it must find a reference to those entities. There exists a globally declared variable *gEntList* which it can use to find the entities it needs. The problem is when to look for the entities. A distinction is made between creating an instance of an entity in source code and spawning it. When creating an instance, the constructor is called, default values are assigned and the values assigned the entity from Hammer are set. Once instantiated, the entity is **Spawned** into the world by calling the **Spawn** function. This places the entity into the world and finishes last initialization based on the values assigned to it from Hammer.

Only once all entities have been properly spawned does it make sense to enumerate the *gEntList* list to search for other entities. The Brainbug uses the Think System of the Source engine to grab references to other entities. An entity can define multiple think functions, one of which can be set to the current think function. The entity can choose a current think function and tell the Source engine when it wants the function called. Most recurring logic of entities is placed within these think functions. The ability to change which think function to use allows for splitting logic into distinct parts.

The Brainbug has four think functions, representing the different states it can be in. When spawned it will set the think function to *FirstThink* and have the engine call it within a tenth of a second into the game. Think functions are called only when all entities have already been spawned which is what we need. Within *FirstThink* the Brainbug will find the references it needs and select the first end sp2_bugpoi the bugs have to reach. Once the FirstThink is over, the next think function is *RoundThink*.

In *RoundThink*, the Brainbug selects the next round. If there are no more rounds the player has won, otherwise the *NewRound* output is fired and if the sp2_round calls for spawning of turrets, the turrets are spawned. Finally, the Brainbug swithches to *SpawnThink*.

The Brainbug has a variable called *SpawnFrequency* which is the interval between *SpawnThink* calls. If the current round has any more bugs to spawn, a bug is spawned

at a random start sp2_bugpoi, otherwise the think function is reverted to *RoundThink* with the delay defined by the current sp2_round.

The main feature of the Brainbug is the ability to spawn bugs, which is the most non-trivial part of the Brainbug:

Listing 7.6: Brainbug SpawnBug implementation

```
// Spawns a bug at a spawn position.
// isRed: is a red bug
// shouldRun should the bug run
void BrainBug::MakeBug(bool isRed, bool shouldRun)
{
    CBugNPC *pent = (CBugNPC *)CreateEntityByName( "sp2_bug" );

    if ( !pent )
    {
        Warning("NULL Entity in BrainBug!\n" );
        return;
    }

    // Give the bug a reference to the brainbug.
    pent->m_eBRAAIN = this;
    pent->m_bCanRun = shouldRun;
    pent->m_nSkin = isRed ? 1 : 0;


    if(isRed)
    {
        // if this bug is red, it should have resistance toward turrets
        // it should also weigh player damage higher than turret damage when path finding
        pent->m_vecArmorColor = Vector(1, currentRound->m_flSpecialsTurretResistance, 0);
        pent->m_vecCostFactor = Vector(currentRound->m_flSpecialsPlayerAvoidance,1,0);
    }
    else
    {
        pent->m_vecArmorColor = Vector(1,1,0);
        pent->m_vecCostFactor = Vector(1,1,0);
    }

    // Set the bugs target Point of Interest
    pent->SetTargetPOI( targetPOI );

    Vector vPos;
    BugPOI* spawnPOI = SelectSpawnPOI();

    // Find somewhere for the bug to spawn. Try within a radius.
    if ( !CAI_BaseNPC::FindSpotForNPCInRadius( &vPos, spawnPOI->GetAbsOrigin(), pent, 0 ) )
    {
        CAI_BaseNPC::FindSpotForNPCInRadius( &vPos, spawnPOI->GetAbsOrigin(), pent, spawnPOI->
            m_flSpawnRadius );
    }

    // Set the bugs position to the result
    pent->SetAbsOrigin( vPos );

    // Strip pitch and roll from the spawner's angles. Pass only yaw to the spawned NPC.
    QAngle angles = GetAbsAngles();
    angles.x = 0.0;
    angles.z = 0.0;
    pent->SetAbsAngles( angles );

    // Set spawn flags!
    pent->AddSpawnFlags( SF_NPC_FALL_TO_GROUND );
    pent->AddSpawnFlags( SF_NPC_FADE_CORPSE );

    // Spawn!
    ChildPreSpawn( pent );
    DispatchSpawn( pent );
    pent->SetOwnerEntity( this );
    DispatchActivate( pent );
```

```
    ChildPostSpawn( pent );

    // count this NPC
    m_nLiveChildren++;
}
```

7.5. The Bug NPC

The bugs are NPCs and therefore the sp2_bug class specializes the existing *CBaseNPC* class, which implements shared logic for NPCs in Source. This gives the bugs the ability to attack an enemy without any more changes. By declaring the player an enemy of the bugs in a default relationships table found in the Source SDK the bug will attack the player when it spots him.

7.5.1. Schedules

To change the default behavior we implemented new schedules (see section 4.2.4 on page 22 for details) for the bugs to use:

Listing 7.7: The sp2_bug schedules, tasks and conditions

```
// New Tasks
enum
{
    // Find a route to the current goal of the bug
    TASK_FIND_BUG_GOAL = LAST_SHARED_TASK,
    // Once burrowed, stay there until woken
    TASK_STAY_BURROWED
};

// New Conditions
enum
{
    // Did the bug reach its goal?
    COND_BUG_REACHED_GOAL = LAST_SHARED_CONDITION,
};

// Schedules
DEFINE_SCHEDULE
(
    SCHED_BUG_UNBURROW,
    "   Tasks"
    "       TASK_STOP_MOVING                0"
    // Play an animation defined on the model
    "       TASK_PLAY_SEQUENCE      ACTIVITY:ACT_ANTLION_BURROW_OUT"
    ""
    "   Interrupts"
    "       COND_TASK_FAILED"
)
DEFINE_SCHEDULE
(
    SCHED_BUG_BURROW,
    "   Tasks"
    "       TASK_STOP_MOVING                0"
    // Play an animation defined on the model
    "       TASK_PLAY_SEQUENCE      ACTIVITY:ACT_ANTLION_BURROW_IN"
    "       TASK_STAY_BURROWED      0"
    ""
    "   Interrupts"
    "       COND_TASK_FAILED"
)

// Once burrowed, stay there!
DEFINE_SCHEDULE
(
```

```
    SCHED_BUG_BURROWED,
    "   Tasks"
    "       TASK_STAY_BURROWED  0"
    ""
    "   Interrupts"
    "       COND_TASK_FAILED"
)

DEFINE_SCHEDULE
(
    SCHED_RUN_TO_GOAL,
    "   Tasks"
    "       TASK_SET_TOLERANCE_DISTANCE     48"
    "       TASK_SET_ROUTE_SEARCH_TIME      3"
    "       TASK_FIND_BUG_GOAL  0"
    "       TASK_RUN_PATH                   0"
    "       TASK_WAIT_FOR_MOVEMENT          0"
    ""
    "   Interrupts"
    "       COND_NEW_ENEMY"
    "       COND_LIGHT_DAMAGE"
    "       COND_HEAVY_DAMAGE"
    "       COND_HEAR_PLAYER"
    "       COND_HEAR_DANGER"
)
// Omitted: SCHED_WALK_TO_GOAL, identical to SCHED_RUN_TO_GOAL
//                  except that it uses TASK_WALK_PATH.
```

With these extra schedules all that is needed is to override the default schedule selection:

Listing 7.8: Schedule Selection for the Bug

```
int CBugNPC::SelectSchedule( )
{
    // If the bug already reached its goal
    if(HasCondition(COND_BUG_REACHED_GOAL))
    {
        if(!m_bIsBurrowed)
            return SCHED_BUG_BURROW; // Begin burrowing
        else
        {
            // Destroy this bug npc entity
            RemoveDeferred();
            if(m_eBRAAIN != NULL && m_eTargetPOI != NULL)
            {
                // Inform the BrainBug (m_eBRAAIN) of this bug's success
                m_eBRAAIN->ReachedPOI(this, m_eTargetPOI);
                m_eTargetPOI = NULL;
            }
            // stay burrowed (Does not matter as bug will be despawned)
            return SCHED_BUG_BURROWED;
        }
    }

    // If the bug already knows about the player
    // and it hates the player more than 20 (this will be elaborated later).
    if(GetEnemy() != NULL && FindOrAddEntityRelationship( GetEnemy() )->priority > 20)
    {
            // Default to the combat logic of BaseNPC.
            return BaseClass::SelectSchedule();
    }

    // Otherwise advance toward the objective location (GOAL)!
    return m_bCanRun ? SCHED_RUN_TO_GOAL : SCHED_WALK_TO_GOAL;
}
```

The final decision making logic implemented is the gathering of our condition and the execution of our custom tasks:

Listing 7.9: Sp2_bug gathering conditions and start of custom tasks

```
void CBugNPC::GatherConditions()
{
    // If within the target Point of Interests radius.
    if(m_eTargetPOI && m_eTargetPOI->GetAbsOrigin().DistTo(GetAbsOrigin()) < m_eTargetPOI->
        m_flSpawnRadius  )
        SetCondition( COND_BUG_REACHED_GOAL );

    // Call base
    BaseClass::GatherConditions();
}

void CBugNPC::StartTask( const Task_t *pTask )
{
    switch ( pTask->iTask )
    {
    case TASK_FIND_BUG_GOAL:
        {
            // Find a route to this bugs target PoI
            Vector v = m_eTargetPOI->GetAbsOrigin();

            if (!GetNavigator()->SetGoal( v ))
                TaskFail(FAIL_NO_ROUTE);
            break;
        }
    case TASK_STAY_BURROWED:
        // Don't draw the bug if burrowed!
        m_bIsBurrowed = true;
        AddEffects( EF_NODRAW );
        TaskComplete();
        break;

    default:
        // Fall back to base implementation
        BaseClass::StartTask( pTask );
        break;
    }
}
```

That covers all of the decision making for the bugs. Now the bug will move toward the goal until interrupted and aggravated to a specific point after which it will instead attack the player.

7.5.2. Taking Damage

To model the aggravation of bugs toward the player, a simple aggression system was implemented. All an NPC's enemies have a **priority** assigned to them. This is used to pick which enemy to attack in BaseNPC. Previously we used this priority to decide whether or not to begin attacking at all. When a bug sees the player it will remember him as an enemy, the priority to attack him should increase if the bug is attacked or if a nearby bug is attacked by the player.

When attacked the bug should also update the danger level of the nearest pathfinding node. To modify the way an NPC takes damage the *OnTakeDamage_Alive* is overridden:

Listing 7.10: Code for bugs taking damage

```
int CBugNPC::OnTakeDamage_Alive( const CTakeDamageInfo &info )
{
    // The attacker (turret or player) has a vector assigned deciding what type of damage to deal
    // Turret DamageColor is (0,1,0), player is (1,0,0)
    // Damage is represented as vectors, which can also be used to represent colors
    // this allows us to have different damage types and refer to them easily.
```

```
    Vector coloredDamage = info.GetAttacker()->GetDamageColor() * info.GetDamage();

    // Red bugs take less damage from turrets
    // meaning the y in m_vecArmorColor is smaller than 1
    Vector dealtDamage = coloredDamage * m_vecArmorColor;

    // Recalculate damage:
    CTakeDamageInfo newInfo = info;
    newInfo.SetDamage((dealtDamage.x + dealtDamage.y + dealtDamage.z));

    // Increase the priority of the attacker by the amount of damage taken
    FindOrAddEntityRelationship( info.GetAttacker() )->priority += newInfo.GetDamage();

    // Find the nearest pathfinding node
    CAI_Network*        ai = g_pBigAINet;
    int nodeId = ai->NearestNodeToPoint(this->GetAbsOrigin(), false);

    // Get the danger level weight system.
    CNodeWeight* weighter = g_pAINetworkManager->m_pNodeWeight;

    if(nodeId >= 0 && nodeId < weighter->Count())
    {
        Vector dangerChange = coloredDamage / sp2_bug_initial_health.GetFloat();

        // Maximum register 1 bug death each time damage is dealt
        dangerChange.x = min(dangerChange.x, 1);
        dangerChange.y = min(dangerChange.y, 1);
        dangerChange.z = min(dangerChange.z, 1);

        // Update the danger level
        weighter->ChangeWeight(nodeId, dangerChange);

        float range = sp2_splash_damage_range.GetFloat();

        // Give neighbour nodes damage according to their distance to the current node.
        CAI_Node* node =  ai->GetNode(nodeId);
        for(int i=0;i<node->NumLinks();i++)
        {
            CAI_Link* link =  node->GetLinkByIndex(i);
            CAI_Node* destNode = ai->GetNode(link->m_iDestID);

            float dist = (node->GetOrigin() - destNode->GetOrigin()).Length();
            if(dist < range)
            {
                float distFactor = 1 - (dist / range);
                weighter->ChangeWeight(link->m_iDestID, dangerChange * distFactor);
            }
        }
    }
    return BaseClass::OnTakeDamage_Alive(newInfo);
}
```

The bugs should also become aggravated if their friends are attacked. When an NPC is attacked, the BaseNPC logic will send a message to nearby friendly NPCs. We intercept this and add aggravation on friendly bugs:

Listing 7.11: Nearby friend bugs damaged.

```
void CBugNPC::OnFriendDamaged( CBaseCombatCharacter *pSquadmate, const CTakeDamageInfo &info )
{
    // If the bug does not already hate the player. Make it hate!
    Relationship_t* r = FindOrAddEntityRelationship( info.GetAttacker() );
    // Add aggravation.
    r->priority += info.GetDamage() * 0.75;

    BaseClass::OnFriendDamaged( pSquadmate, info );
}
```

7.5.3. Advancing

To use the danger level system the bug needs to add additional cost to its pathfinding algorithm based on the danger on the nodes. Source already has a system in place to add additional cost to some node edges based on the type of movement required to travel along the edge. For example edges can be marked as swim or jump edges. We modified this system to add additional cost to edges based on the danger level associated with the destination node of an edge:

Listing 7.12: Calculating the cost of traveling along an edge.

```
bool CBugNPC::MovementCost(int movetype, const Vector &vecStart, const Vector &vecEnd, float *
    pCost, CAI_Node* start, CAI_Node* end)
{
    if(end != NULL && end->GetId() < g_pAINetworkManager->m_pNodeWeight->Count())
    {
        // Get the weight of the end node from the node weighting system.
        Vector weight = g_pAINetworkManager->m_pNodeWeight->GetNodeWeight(end->GetId());

        // Apply a factor for weighing some damage higher than others
        // This is used for Red bugs to avoid places where the player has dealt damage
        weight *= m_vecCostFactor;

        // Calculate the final danger level. Cap to 10 to avoid pathfinding acting up.
        float additional = min(weight.x + weight.y + weight.z, sp2_danger_level_cap.GetFloat());

        // Add the additional cost using the danger cost factor.
        *pCost += additional * sp2_danger_cost.GetFloat();
    }
    return true;
}
```

7.6. Map Creation

The creation of the map is done using the level editor Hammer, as described in section 3.2 on page 11. The map will adhere to the layout described in Map Design 6.3 on page 40. The map should resemble terrain found in an arid valley in between impassable rocks. Hammer's basic world geometry are blocks, wedges and triangles. All these are very hard edged making them unsuitable for creating environments which need to resemble natural formations, such as sand, rocks or dirt. Instead of using basic geometry to create terrain, a displacement option can be applied to the geometry. This feature turns the blocky world geometry into a displacement surface, which can then be shaped or molded into land formations, as seen in figure 7.2.

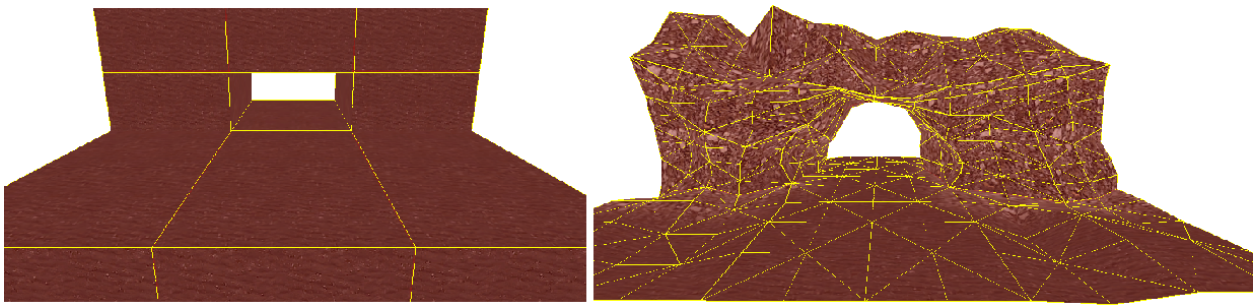


Figure 7.2.: Image showing basic world geometry on the left side, and the same geometry with displacement on the right creating the semblance of cave geometry.

The displacement option is used to create the overall layout of the terrain. As the

map is mostly dirt and rock formations, it needs some sand bunkers to make it seem reasonable that the bugs could emerge from there. The sand bunkers are used as places where the bugs can emerge from and burrow into. Basic block geometry can be used to represent a sand bunker, when it is applied a proper sand texture, and placed in a location such that it looks right. An example of this can be seen in figure 7.3.

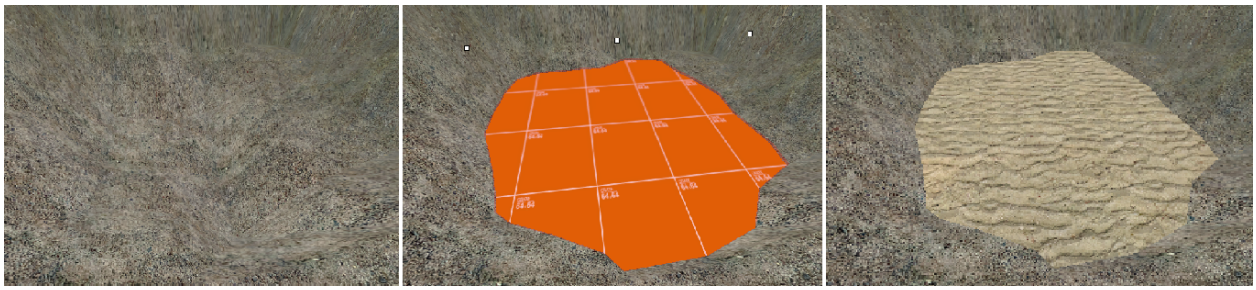


Figure 7.3.: Image showing a sand bunker created in three steps. Step one is to find a suitable location or create one by sculpting the displacement mesh. Step two is to create some geometry in the location. Step three is to change the texture to a fitting sand texture.

Lastly the level is surrounded by a hollow box which has a skybox texture applied. This seals off the interior of the level from the void and prevents any leaks from occurring. Adding the sky texture gives the illusion of a real sky; when the player moves, the sky in the horizon moves.

After the overall layout of the map is complete, the logic should be introduced, otherwise the map will be static and uninteresting. Entities are used to add logic. The map is created using our own entities as well as those already present in Source. The first entity to be added is the sp2_brainbug, which is needed to start the game. This entity can be placed anywhere in the map as it is not used in conjunction with a specific location.

For the bugs to be able to spawn, the sp2_bugpoi entity is used. The sp2_bugpoi is location dependent, meaning that the sp2_bug will spawn in the entity's location.

The map also needs a Nodegraph which is created by inserting info_node entities into the map at appropriate locations. The Nodegraph is necessary for pathfinding.

Eleven sp2_round entities were added. Table 7.1 on the following page shows the values associated with each entity for a particular round. The round parameters have been determined during internal testing.

When a new round is initiated an env_message entity for the corresponding sp2_round is triggered. This will display a message accordingly to what was stated in Map Design about Messages in section 6.3.2.1 on page 42.

Figure 7.4 on the following page shows the map while it is being worked on in Hammer. The 3D view displays multiple info_node entities which are the flat yellow boxes. Also seen is one of four red sp2_bugpoi entities, each placed in the middle of a sand bunker. In the background the displacement terrain is visible.

The three 2D views show the map from the sides and the top. Most of the logic entities can be seen in the bottom rightmost view. Although only a single sp2_round and env_message is visible in the image there is a pair for each round totaling eleven pairs handling the round information.

Round	Yellow Bugs	Running Yellow Bugs (%)	Red Bugs	Turrets
1	30	0	0	1
2	40	0	2	0
3	50	0	4	1
4	60	20	6	0
5	70	25	8	1
6	80	30	10	0
7	90	35	12	1
8	100	38	20	0
9	100	40	30	1
10	150	40	40	0
11	200	50	50	1

Table 7.1.: Round Parameters for this map.

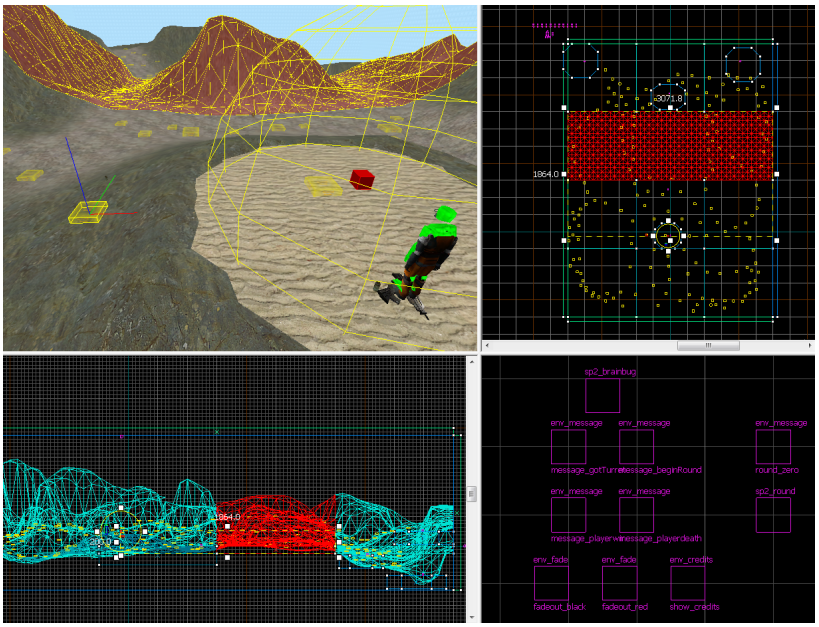


Figure 7.4.: Image showing the interface of Hammer. Displacement geometry, a path finding node and an sp2_bugpoi entity have been highlighted.

7.7. Smart Bugs Screenshots

This section contains images from the running game. The images are figures 7.5, 7.6, 7.7 and 7.8.

Figure 7.5.: Player defending the objective location, which has almost been overrun.

Figure 7.6.: A red bug burrowing into the objective location.

Figure 7.7.: Turret and player attacking an incoming wave of bugs.

Figure 7.8.: It is apparent that the bug on the left in this screenshot has been killed by a turret, because it did not explode when killed.

CHAPTER

8

Test and Evaluation

This chapter deals with test and evaluation of the implementation described in chapter 7 on page 44. Section 8.1 contains tests of the implemented algorithms for the node weight system. Section 8.2 on page 74 describes a Playtest involving users. Finally, section 8.3 on page 78 describe the goals for the next iteration of development, based on the results.

8.1. Test of Algorithms

In section 6.2.1 on page 33 multiple algorithms for removing danger over time on the pathfinding nodes in the game were described. This section analyzes and compares the different algorithms and finds the most suitable one for the game. Furthermore, it describes how the algorithm is used in the game and how it affects gameplay.

8.1.1. Finding the best algorithm

We have to determine which of the four algorithms defined in section 6.2.1 on page 33 is the best fit for the game. In order to do this we create a simple test scenario which allows us to compare the different danger removal systems. The test scenario will be a simulation of what happens in the game: For each algorithm we will have a Brainbug send bugs down a single path which has a single weighted node. All bugs which get to this node are instantly killed, adding a value of 1 dead bug to the node weight. By adjusting how many bugs the Brainbug sends out and how many updates there are between waves, we can test different characteristics of the algorithms.

The following settings will be used for the different algorithms:

Algorithm I A fade-per-update value of 1 will be used.

Algorithm II A decay factor of 0.9 will be used.

Algorithm III A sample size of 10 will be used.

Algorithm IV A merging factor of 0.9 will be used.

In order to make the output from the algorithms comparable, the output from Algorithm II will be scaled by a factor of $1 - 0.9 = 0.1$.

Three test are performed, each eliminating algorithms which are unsuitable for the game. Finally, only one remains:

Test 1 Handling a constant flow of bugs.

Test 2 Handling waves of bugs.

Test 3 Handling variable wave sizes.

8.1.1.1. Test 1: Constant Flow of Bugs

This test examines how the algorithms handle a constant flow of bugs. First we test how the weights change when one bug is sent out for every update. The result is shown in figure 8.1. All algorithms manage to detect what we expect them to: A danger level of 1 due to the fact that one bug is killed at the node in every update. We can note that Algorithm I instantly reaches the value 1, while the others approach it differently. Algorithm III increases rapidly in the beginning but is later overtaken by Algorithm II, while Algorithm IV is slower than both of them. The three last algorithms all behave similarly but Algorithm IV is a bit slower in the beginning.

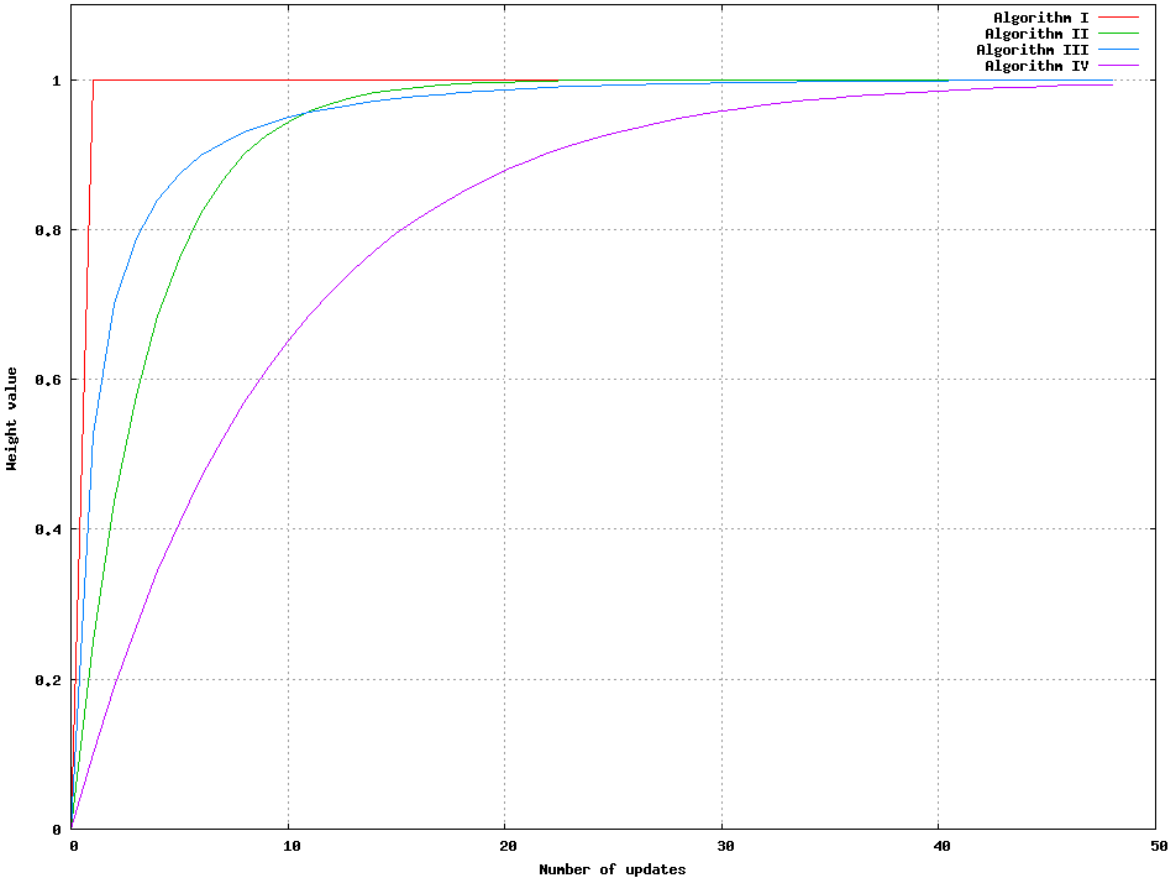


Figure 8.1.: Test 1 - Weights dispatching and killing one bug every update.

The rate of bugs is now changed such that three bugs are dispatched each update. The result of this is shown in figure 8.2 on the next page. This highlights a problem with Algorithm I: It will continuously increase the weight, giving the weight a high resistance to change. This can be corrected by changing the algorithm's fade-per-update setting, but it is difficult to find a value which would work for all nodes at all times in the game. Therefore Algorithm I is discarded as a candidate and we concentrate on testing the last three algorithms in the next two tests.

8.1.1.2. Test 2: Waves of Bugs

We now test how well the algorithms handle waves of bugs. We start with a test in which 10 bugs are dispatched every 10 updates. The result can be seen in figure 8.3 on page 64. We can learn two things from this:

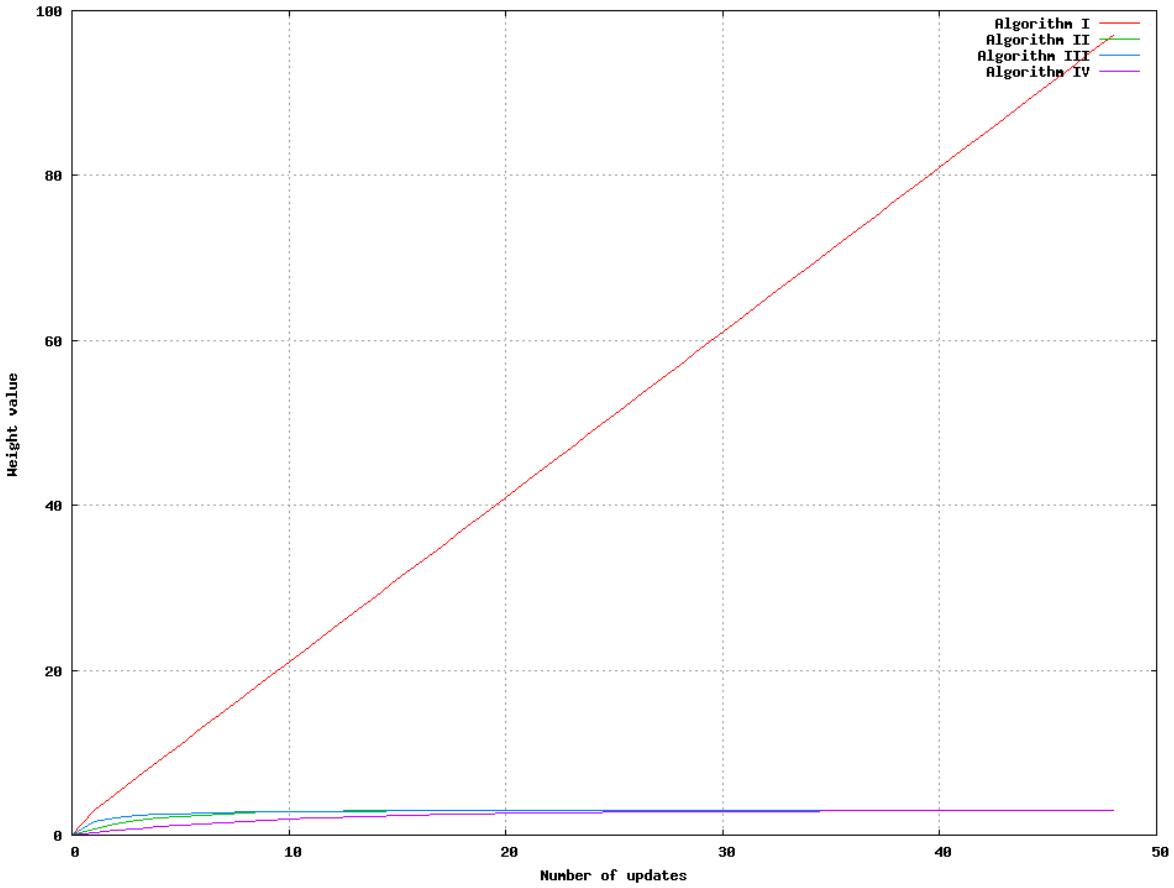


Figure 8.2.: Test 1 - Weights dispatching and killing three bugs every update.

Algorithm II has a larger value range than the two other algorithms. It has a higher maximum and a lower minimum than the others, however it still does what is expected: Like the two other algorithms it jumps up to a high value whenever a wave occurs and then slowly decreases.

Algorithm III and IV approach each other as the number of updates increases.

This test does not gives us any reason to discard any of the algorithms. They perform as expected although they have different value ranges.

8.1.1.3. Test 3: Variable Wave Sizes

Our algorithm should adjust to differences in the size of waves. If we define a route to no longer be dangerous when the weight on that route drops below a certain threshold we want the number of bugs killed at the node to affect how many updates are needed before the weight's value drops below the threshold. If for example 10 bugs are killed we want the algorithm to take longer to drop below the threshold than if only 5 bugs were killed.

We can test this by introducing a simple Brainbug in our scenario which will dispatch bugs whenever a weight drops below a certain threshold. When it dispatches bugs it will alternate between dispatching 10 and 5 bugs. With a threshold of 0.25 we get the result shown in figure 8.4 on page 65.

The first to notice is that Algorithm III has a huge spike at the beginning of the test. This would be undesirable in the game where the bugs should take some time to learn

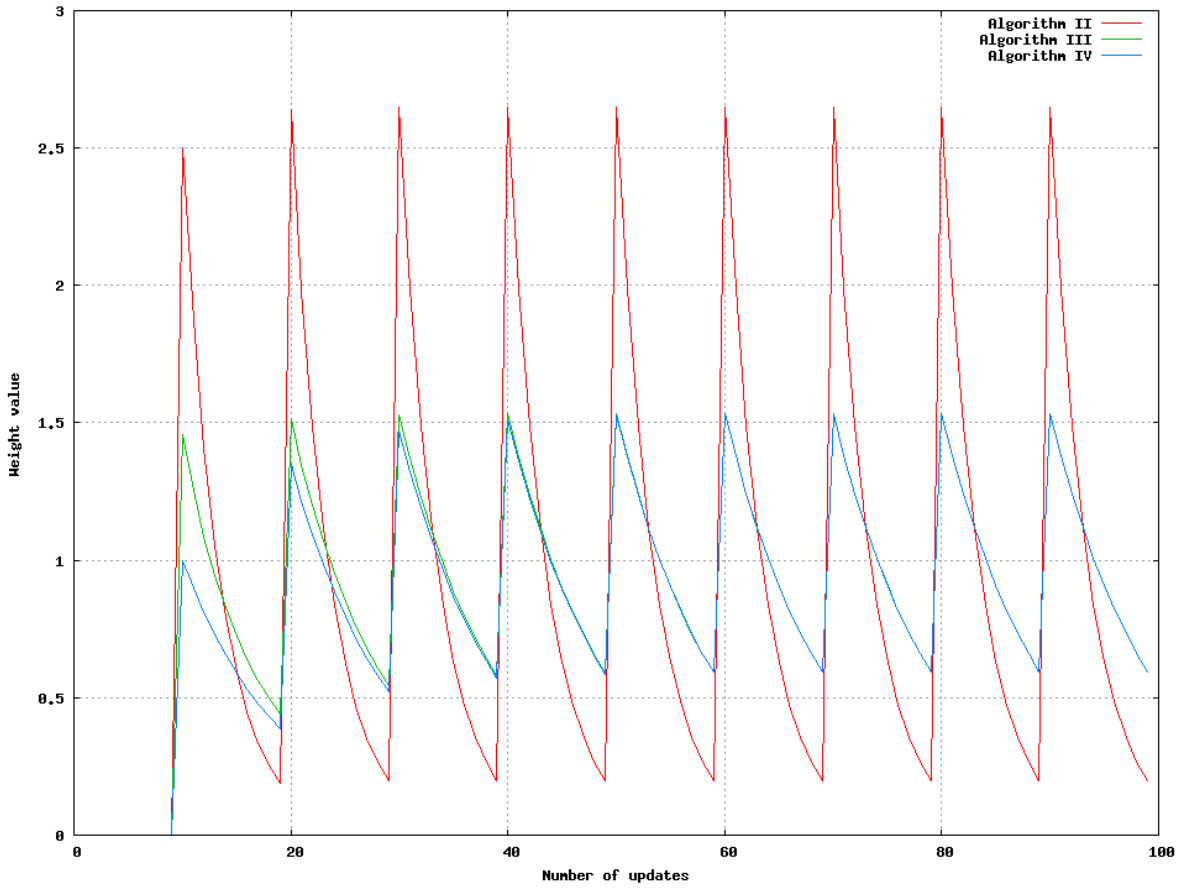


Figure 8.3.: Test 2 - Weights after dispatching and killing 10 bugs every 10 updates.

to follow another path.

Since Algorithm IV starts out with a relatively low weight value and is nearly the same as Algorithm III, it is preferable over Algorithm III. This leaves Algorithm II and IV as candidates. However it is apparent that Algorithm II does not change significantly with changes in the size of the waves. It takes nearly the same time to recover from a 10 bug wave as from a 5 bug wave. Using Algorithm IV there is a noticeable difference, making it the best choice of algorithm to use in the game.

8.1.2. Application on Test Map

To see how this algorithm performs in the game we created a test scenario. We used an adaptation of the map created for the game. In the test map we visualize the routes the bugs would have taken according to the weights on the map. The parameters of the test are:

Algorithm The algorithm chosen for learning is Algorithm IV. The merging factor is **0.05** which corresponds to a sample size of **20** causing the algorithm to take the last 20 updates into account.

Sample Rate In the test scenario the sample rate is **5 seconds**, meaning the algorithm is invoked every **5th** second.

Danger Weight This factor is the distance the danger level on a node corresponds to. It is set to **2400**, which approximately corresponds the average euclidean distance

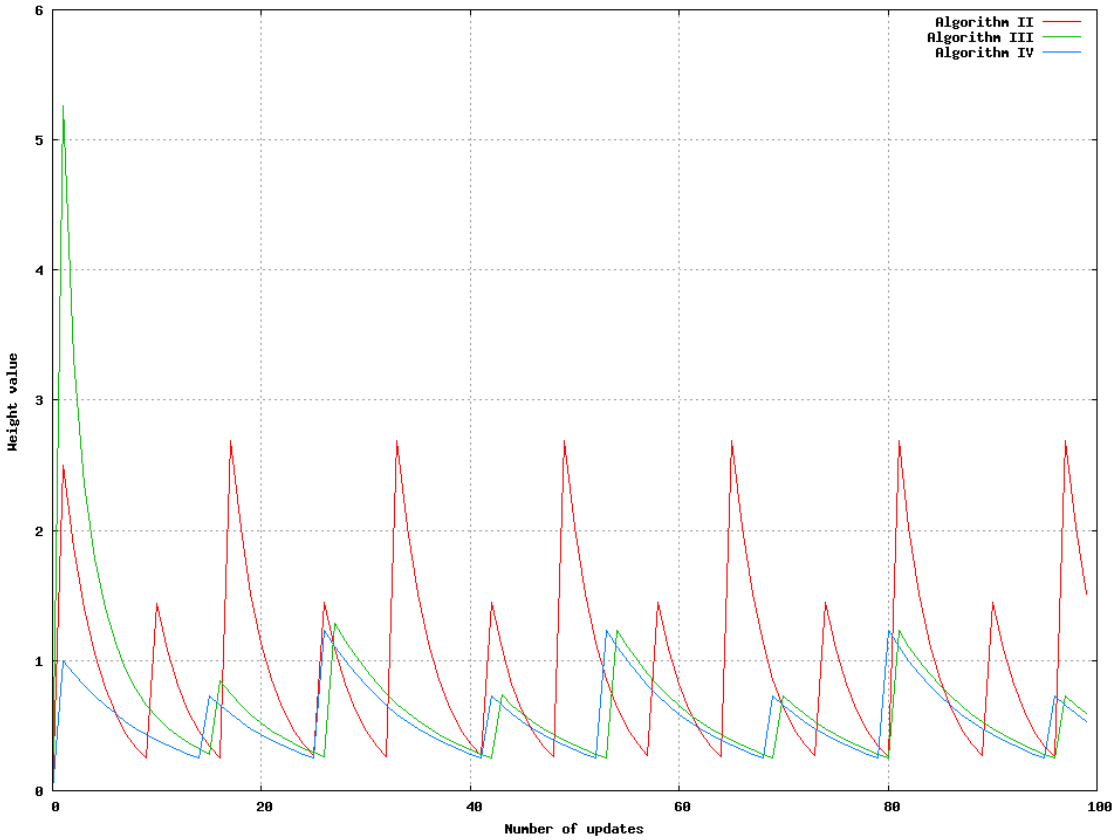


Figure 8.4.: Test 3 - Weights after dispatching alternating waves of 10 and 5 bugs whenever a weight's threshold drops below 0.25.

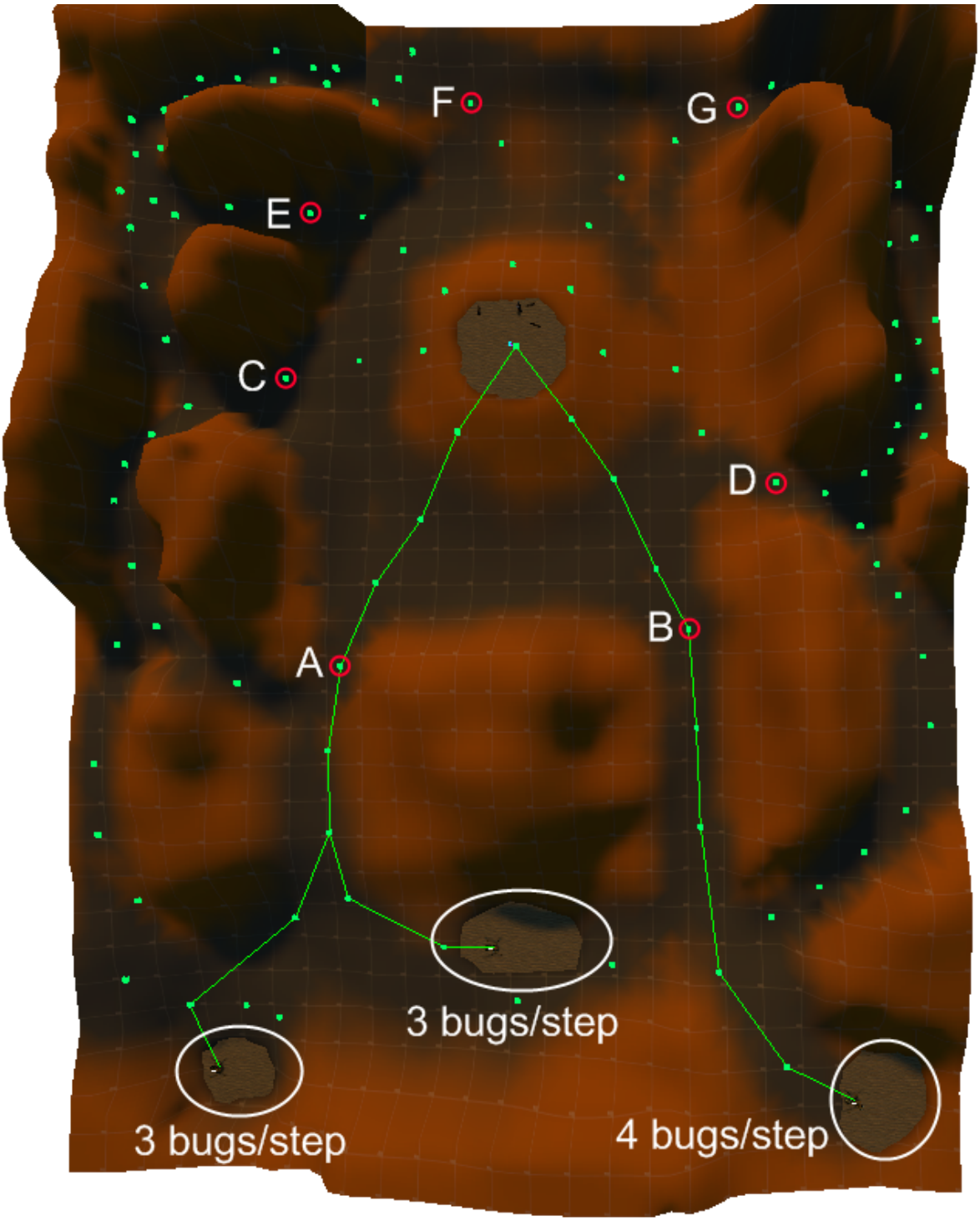
from the spawn points to the goal in the map. This means passing a node which has a danger level of **1.0** corresponds to traversing **2400** units in the game.

Spawn Rate The test map has three different spawn positions and one bug is spawned every half second. This means that 10 bugs are spawned for every update. For the test map we assume that during each sample three bugs are spawned in two spawn locations and four in another.

The test will be performed by using a hypothetical situation: Instead of having the random behavior of an actual game we influence the nodes directly and recalculate the paths the bugs would have taken if in the map. The map has **7** main routes which lead to the center from the three spawn positions, four on the left side of the map and three on the right. We have chosen the first node visible from the center on every route as the nodes on which to register the kills of the bugs. The seven nodes are marked on the figure associated with the initial step on page 66.

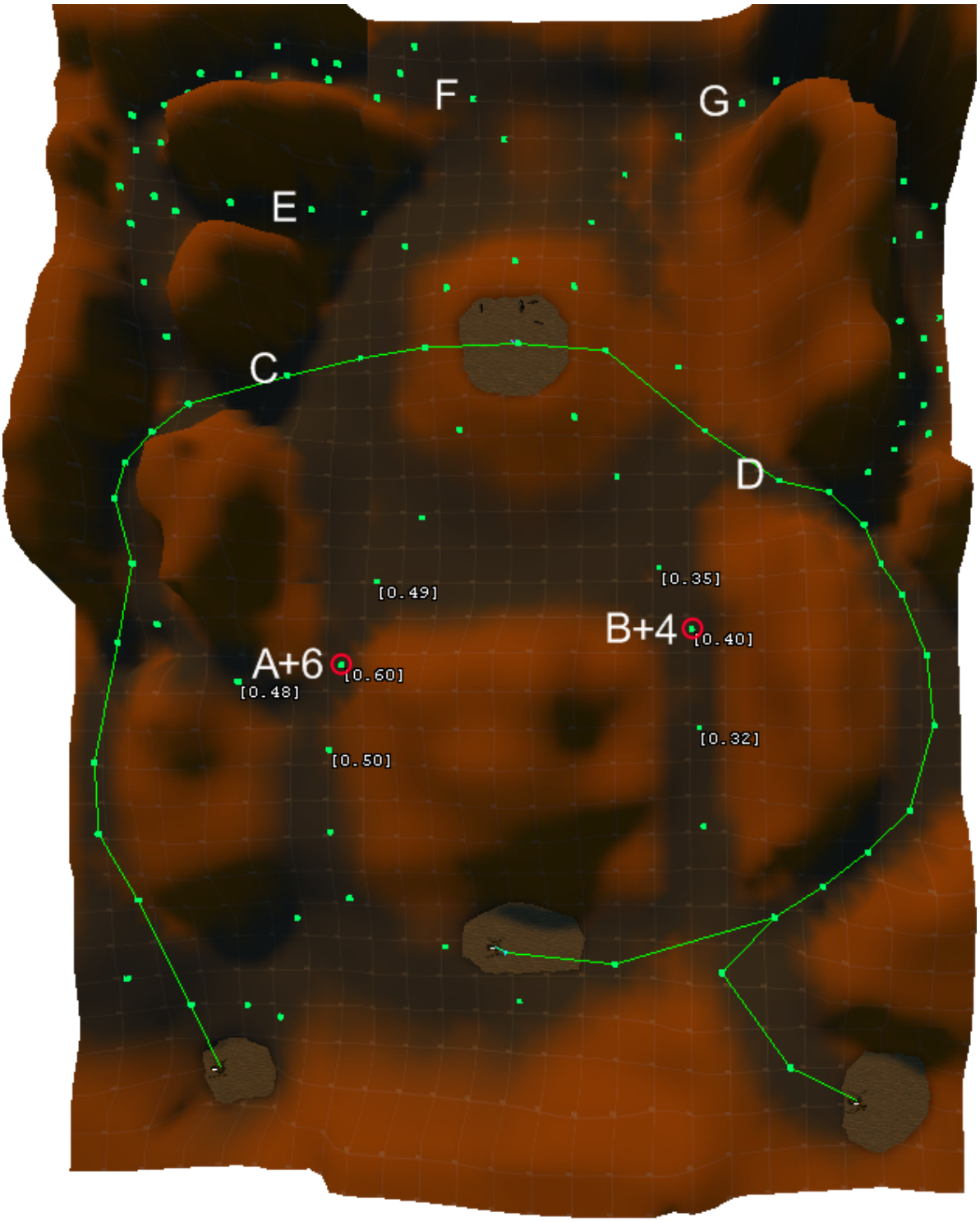
To simulate that a bug was killed we implemented a console command such that pointing somewhere on the map and invoking the command simulates a bug dying at that position, using the same algorithm which would be used if a bug had actually died there. Another command was bound to the update function on the algorithm.

For each sample we simulate the deaths of 10 bugs and run an update to see the effect it has on the path the next 10 bugs will take. We do 5 of these steps and explain the changes.



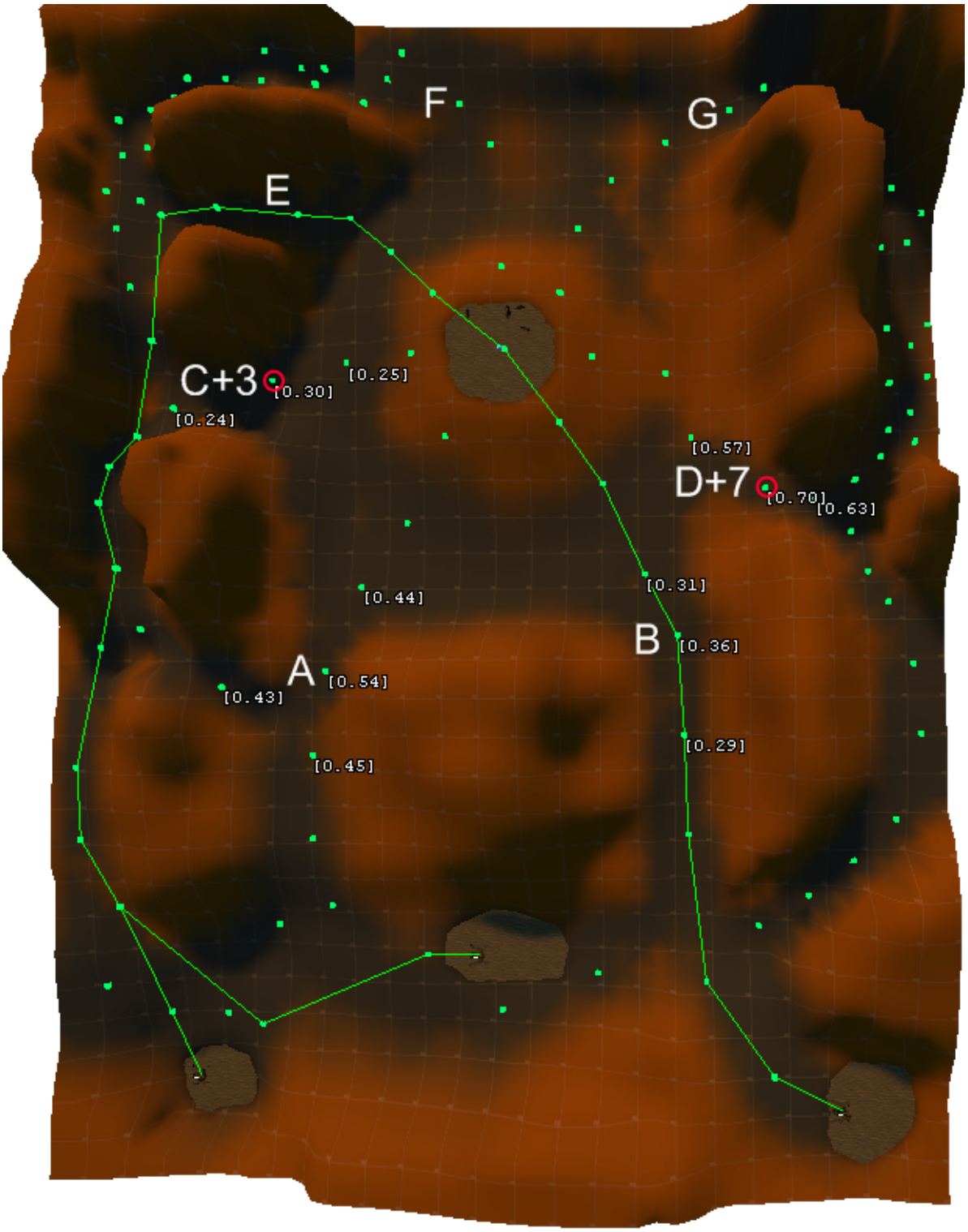
Step 0

This is the initial state. The bugs will take the shortest physical path from their spawn locations to the objective location in the middle of the map. The image also shows the amount of bugs spawned at each spawn location in every step. The green lines are the paths the bugs will take to reach the objective location. The green spots on the map are pathfinding nodes. The nodes marked by letters are where we simulate the kills of bugs. All changes in danger on these nodes during the steps are shown on figure 8.5 on page 73.



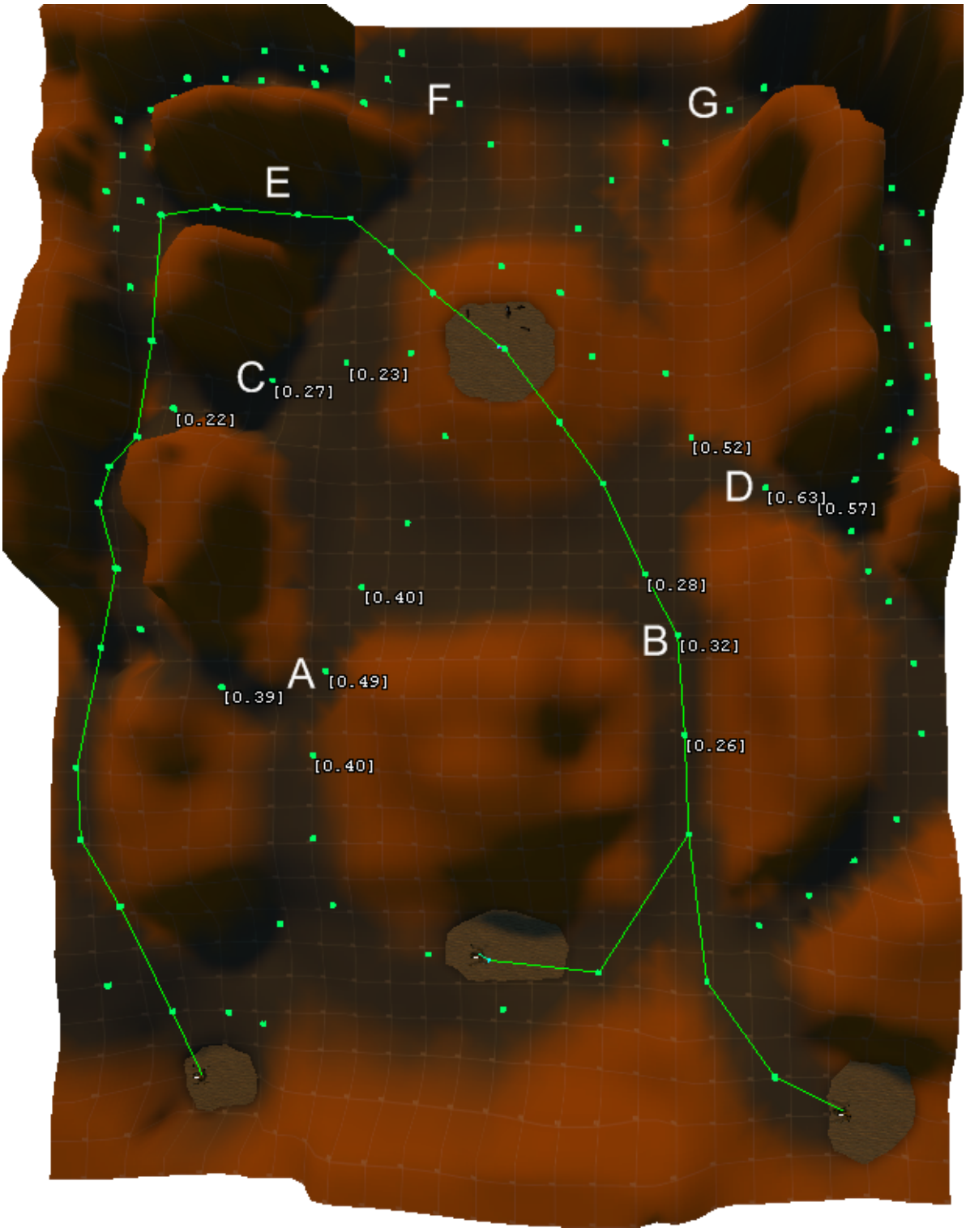
Step 1

In this screenshot ten bugs were killed. The numbers prepended with a plus sign show where the bugs were killed and how many. The numbers in brackets are the danger level at the respective pathfinding nodes.



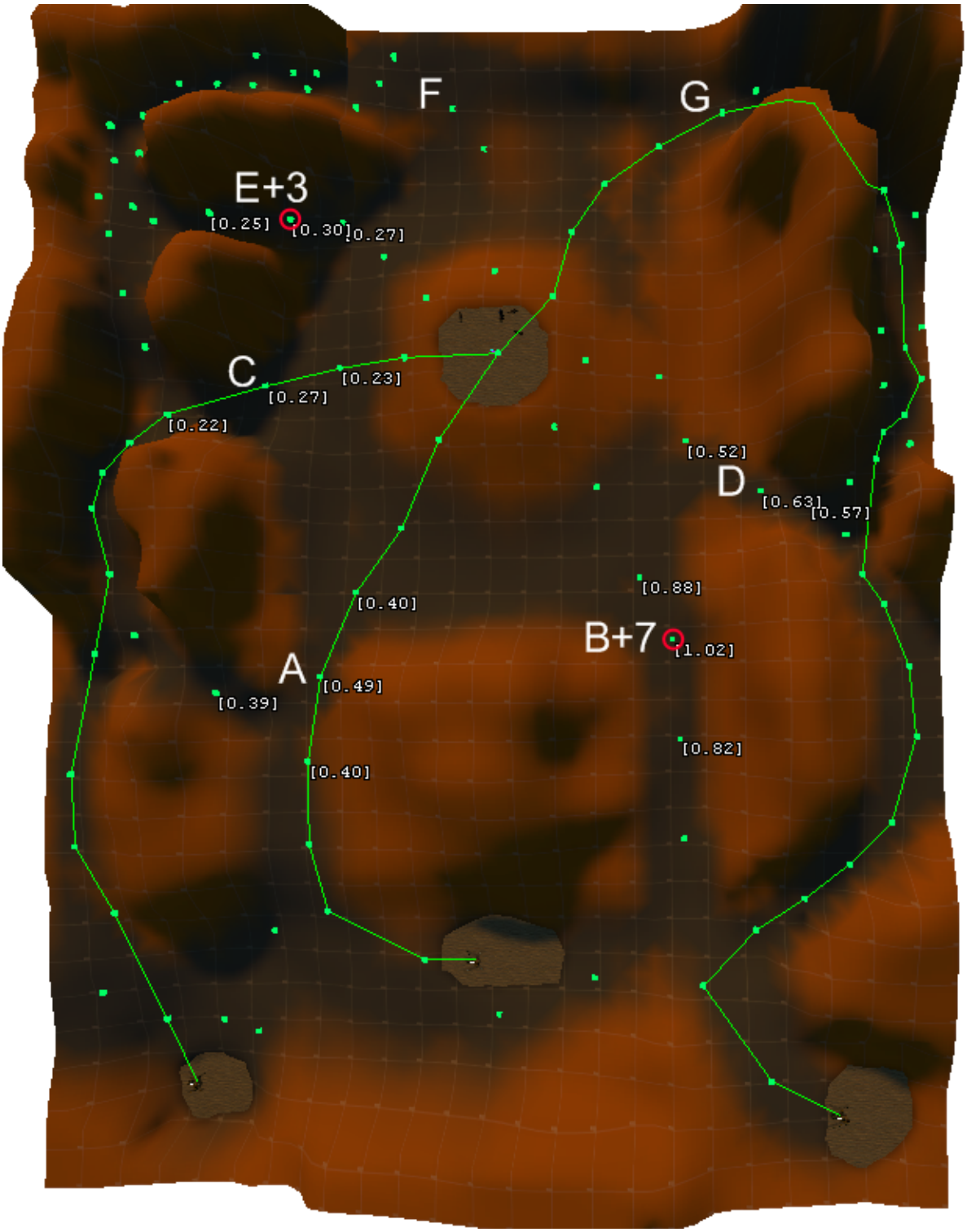
Step 2

After an update and the killing of another 10 bugs. Bugs from the center and left spots go around the goal attacking from behind, which is the intended behavior. The bugs from the right take a shorter route because the danger is not high enough to justify going all the way around.



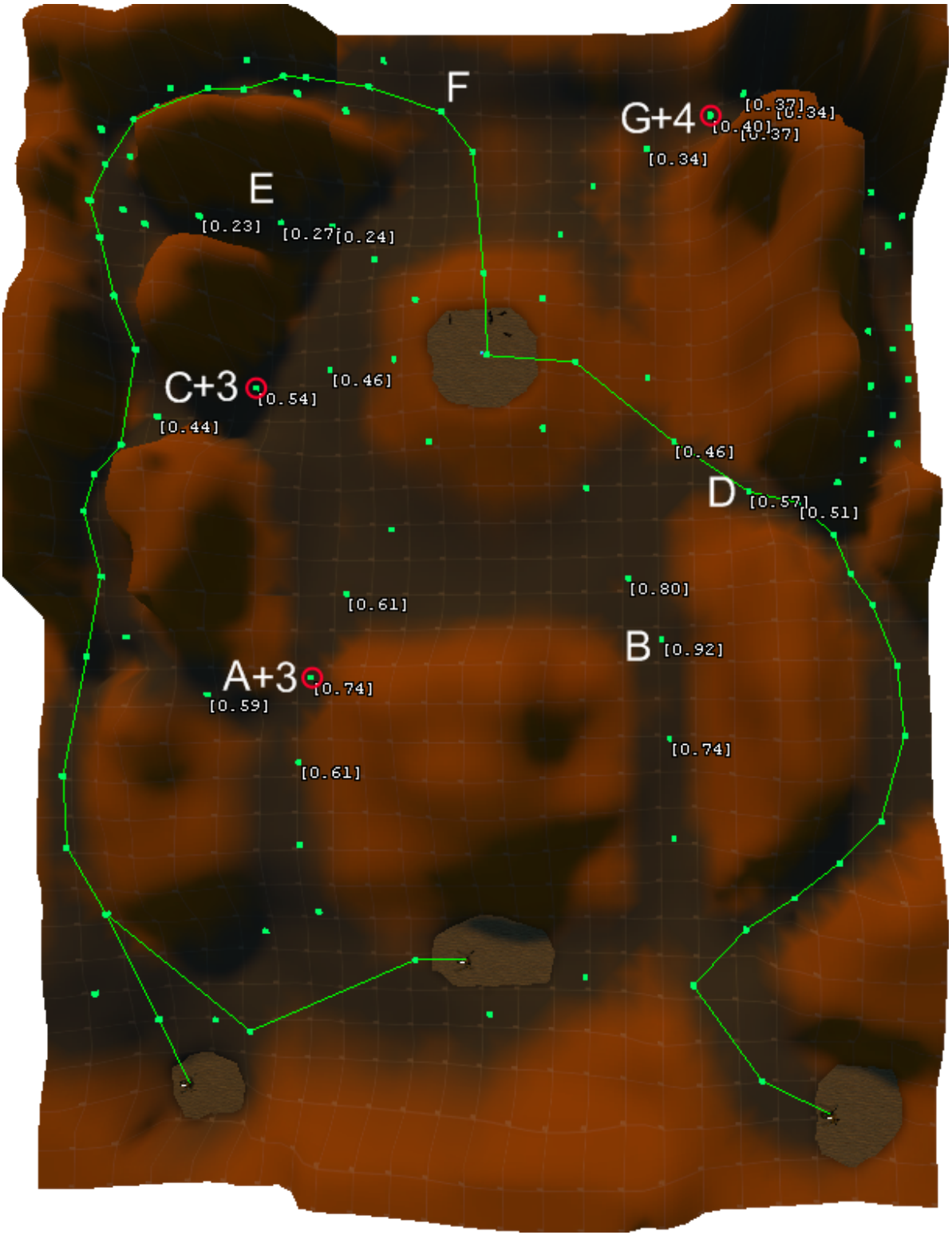
Step 2.5

This is the map immediately after updating the weights from Step 2. The noticeable thing here is that the center bugs have rerouted their path to follow the right bugs instead of the left.



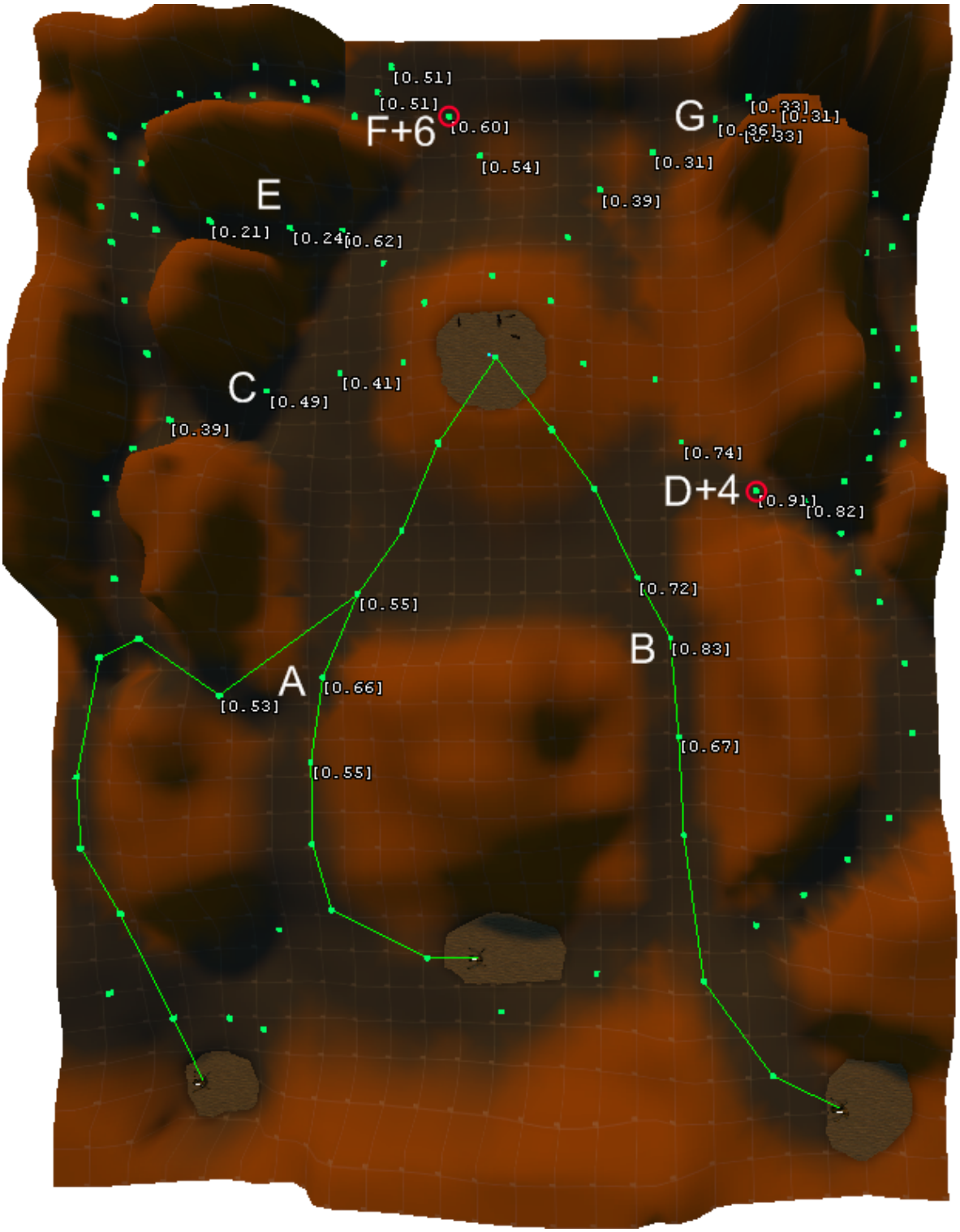
Step 3

The right bugs now go all the way around to attack from behind. The other bugs have reverted to using shorter paths.



Step 4

Because of the immediate danger of the shorter paths, the left and center bugs have found a route all the way around the player. The right bugs have reverted to a shorter path.



Step 5

This final step shows the bugs reverting to something resembling their initial paths. The long routes have become too dangerous and the bugs now attack head on once again.

Test Results

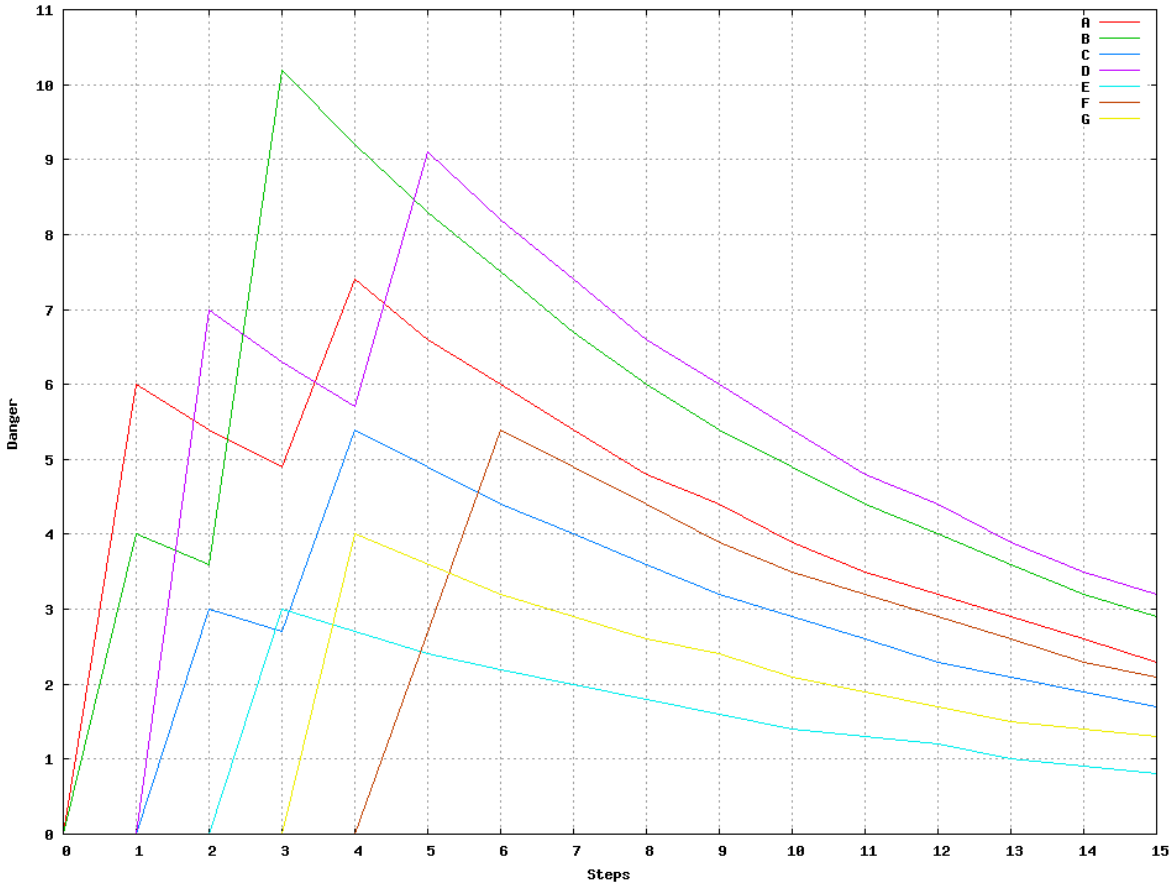


Figure 8.5.: The danger levels on the denoted nodes from step 0. Ten update steps in which no bugs are killed have been added to visualize the fall in danger level over time. The danger levels have been scaled by 10 for readability.

From this simplified test it is apparent that the bugs will avoid dangerous routes. The perceived behavior from the player standing in the middle of the map is that the bugs rarely follow the same path two updates in a row implying that they will attempt to avoid danger. Furthermore, the bugs will return to previous routes when other routes are equally unsuccessful enabling bugs to switch between different routes.

If the sample size is increased, the bugs will react slower to changes and during a longer period of time will learn the player's tendencies, that is the areas in which he successfully defends the goal and which areas he is not paying as much attention to.

Figure 8.6 on the next page shows the total path distance of the seven marked routes during the 5 steps for the bugs spawned in the center spawn location. The total path distance is the actual path distance and danger weight adjustment combined. The danger weight adjustment is calculated by taking every node on the route with a weight and calculating the additional distance using the danger weight. For example in step 5, the screenshot shows the center bug moving through the **A** node, it passes three nodes with weights and so the total path distance is: $(0.55 + 0.66 + 0.55) * 2400 + 2713.06 = 6937.06$ where 2400 is the danger weight and 2713.06 is the actual distance of the route passing through A from the center spawn location.

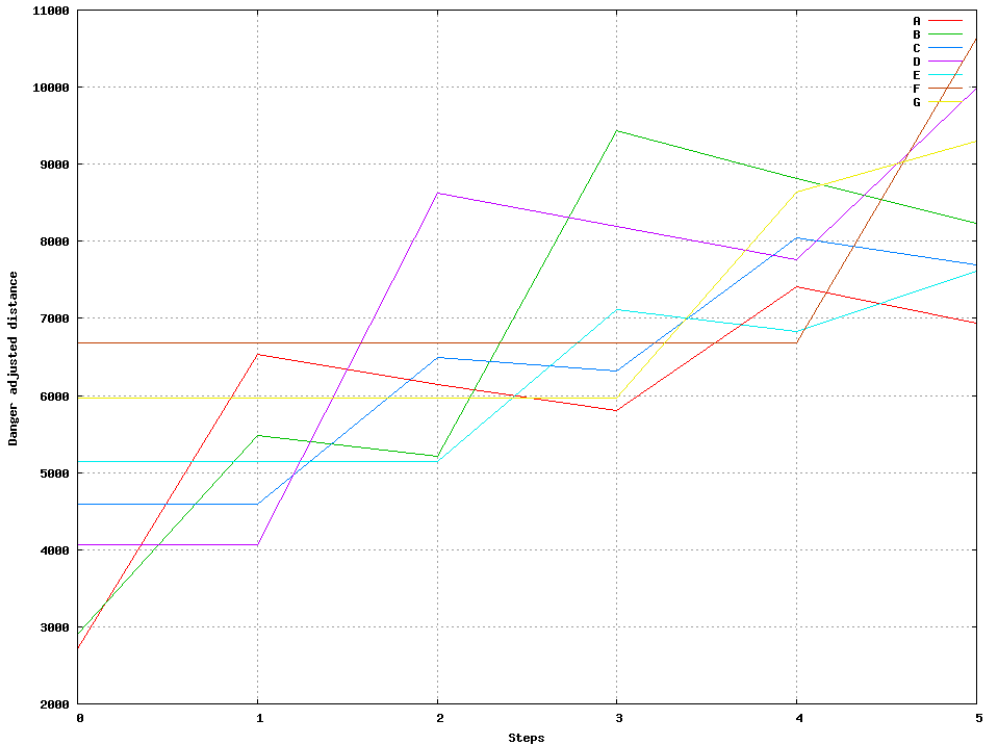


Figure 8.6.: The total path distance of each path passing through a labeled node as perceived by bugs spawned in the center position. The lowest value at each step is the path the center bugs will take that step. These correspond to what is seen on the screenshots of each step.

8.1.3. Test Summary

The first goal of this section was to test the four different algorithms and decide which one was the most suitable for the game. The algorithms were compared using simple test scenarios and elimination was used to choose Algorithm IV as the best algorithm.

In the second section the danger level system was tested and it was found that the bugs can learn to avoid the dangerous routes as well as retry them when enough time has passed to make them seem safe again.

8.2. Evaluation using Playtesting

This section deals with evaluation of our implemented game prototype. The evaluation is done by involving potential end-users. The background of evaluating the users experience of a game was detailed in chapter 5 on page 27.

Section 5.3 on page 30 outlined a number of different evaluation methods. Among these methods we choose Playtesting as our approach. There are three reasons for this. Firstly, Playtesting provides a reasonable amount of data, in proportion to its low cost. Secondly, we are only evaluating the first prototype of a concept, so it does not make sense to do very detailed evaluation on for example Usability. Thirdly, we would like to get some hands-on experience with a method which is actually used in the game industry[2].

The planning of the Playtest is described in 8.2.1. The setup and execution is described

in 8.2.2 on page 77. Finally, the results of the evaluation are presented in 8.2.3 on page 77.

8.2.1. Planning Phase

The background of the Playtest method was described in section 5.3.5 on page 31. Our goal is mainly to get some general feedback on the game concept. The project focus has mostly been on AI-related aspects and learning the Source engine, so we find it relevant to determine if people actually like our game concept.

The prototype is fully functional in terms of the intended game play. Currently, there is only one playable level.

As we intend to have several participants in the evaluation, we need to make sure everyone gets the same information and instructions. To ensure this, everything is put into a hand-out. The intended evaluation procedure is to answer a few demographic questions, read the instructions for the game, play the game and then answer some questions about the game. The hand-out given to the participants can be seen in appendix A on page 83.

The background for the asked questions is presented in the next paragraphs. The questions are essentially designed to cover three areas of interest: The concept, the AI and the gameplay.

Evaluation Subject: The Game Concept

We ask three questions related to the game concept:

- *What is your overall impression of the game?*
  This is asked as a multiple-choice question, with the following choices: "*Overall positive*", "*Fairly positive*", "*Fairly negative*" and "*Overall negative*". The rationale behind this is to get a quantifiable measure of the participant's opinion. This can be used in conjunction with other questions, to identify more complex coherences.

- *What did you like about the game?*
  This is asked as an open question, where the participants are presented with three empty lines to write their answer on. The purpose is to get some feedback on what the participants think about the game in general.

- *What did you dislike about the game?*
  This is asked in the same way as the above question. The purpose is also the same.

In addition to these questions we ask demographic questions about their three favorite games and game genres. The rationale behind this is to have the option of identifying if there is a coherence between game preferences and whether they like our game or not.

We see our game concept as a Shooter type of game, with some tactical elements in form of the bugs, which will try to outsmart the player. To increase the data basis for feedback on the concept, we will evaluate two different variations of the game. The starting point is the bug amounts in table 7.1 on page 58. These numbers were found by testing and tweaking the game ourselves. We use them as one variation of the game, now called "**Many Bugs**".

Now we introduce a new variation of the game, which we will call "**Hard Bugs**". All bugs gets twice as many hit points. The number of yellow bugs and red bugs is reduced by 50%. The percentages of running yellow bugs and turrets stay the same.

The actual game is the same. The "Many Bugs" variation may call for a more action packed game, while "Hard Bugs" aims for more tactically influenced gameplay. By evaluating these two variations we aim to get input on which type of game is more appealing.

As a final question we ask "*What should we work on, in order to improve the game?*", which is stated as a open question. Here we aim for two things: **1)** Getting some new ideas which we may not have thought of earlier. **2)** Identifying tendencies on which game features and development directions are requested.

Evaluation Subject: The Game AI

The AI is a substantial part of our game, so we are interested in feedback on it. As the player is playing against the AI, there is the inevitable issue of matching the strength of the AI against a player with arbitrary gaming skills. We consider this a matter of tweaking constants in the game code. As for example: How many bugs emerge at once? How much damage does the player weapon deal?

The more interesting subject is how our AI performs over time, in terms of proving sufficient resistance to the player. This is relevant as the AI utilizes learning techniques, which the player should experience. We will try to get input in the following manner. We ask the following two questions: *How would you rate the difficulty in the beginning of a game?* and *How would you rate the difficulty towards the end of a game?*. Both questions stated as multiple-choice with the possible choices "Too easy", "Fair" and "Too hard".

We combine this with the two different variations of the game, as mentioned earlier, "Many Bugs" and "Hard Bugs". We hope this approach will provide us with input on how the AI skill is perceived.

We are aware that the perception of the game difficulty might be heavily influenced by the gaming skill of the player. To compensate for this, we ask a demographic question of "*How would you characterize your previous experience with First-Person Shooter types of computer games?*". This is stated as multiple-choice, with the choices "Novice", "Intermediate" and "Advanced". We consider some FPS experience as essential for being able to complete the game. This way we can eventually check the coherence between a perception of high difficulty versus low FPS skills.

Finally, we ask "*What is your impression of the behavior and intelligence of the bugs?*", as an open question. The purpose is to get a general opinion of the AI implementation.

Evaluation Subject: The Gameplay

A final area of interest is the gameplay. Based on our own testing we have discovered a number of gameplay elements which we see as potentially annoying. We would like some concrete input on whether other people perceive this in the same way. We give them the following assignment: "*Please prioritize the following game play elements in order of how much they annoyed you. Prioritize using the numbers from 1 to 5, where 5 is most annoying.*".

As available options we give the following statements:

- *There were too many bugs*
- *There were too few weapons*
- *There were no power-ups*

- *The gameplay was too monotonous*
- *The turrets fell over too easily*

The rationale behind this question is to test whether the users have the same view on sources of irritation as we do. A combination of this and the open questions on likes/dislikes will hopefully provide us with an idea of what works in terms of gameplay.

8.2.2. Execution Phase

16 participants were invited to the Playtest. The test location was a standard university group room. The evaluation setup was designed to mimic a standard gaming environment. This implies a dimmed room and comfortable armchairs. The reason was to make the participants feel comfortable and avoid the feeling of an artificial environment.

The hardware used was two laptops, with external keyboards and mice. One laptop had the "Many Bugs" variation and the other "Hard Bugs". The participants were randomly assigned to a laptop, and were not aware of the two different game variations. The Playtesting was always done with two participants simultaneously. The advantage of this was effective use of time and more comfort for the participants, as they did not have to endure sitting alone.

The use of the hand-out with all necessary information written down made the execution run smoothly. A test leader was always present in the room in case the participants had questions or technical problems. A Playtesting session featuring two participants lasted on average 25 minutes. The entire execution phase was completed in a single day.

8.2.3. Results

This section describes the derived results, based on the analysis of the hand-outs. An extract of the filled out hand-outs can be seen in appendix B on page 89.

The 16 participants were fairly random though with a tendency to be experienced in gaming; 10 participants characterized their previous experience with First-Person Shooter types of games as "Advanced", and 3 on both "Novice" and "Intermediate". This means that while the distribution of the participants leans toward experienced players, there were more than one participant of each experience level. The distribution of results is spread over the range of skill level though it was slightly skewed toward more experienced gamers.

The difficulty level of the game must thus be assessed with this in mind: The participants rated the gaming experience positively, with all sixteen participants rating it better than average: 4 rating it "Overall positive" and 12 rating it "Fairly positive"[1]. On this basis the game prototype can be said to have been received positively by all participants.

The results show that the Playability and Replayability of the game is low. Some participants indicate that the game gets monotonous and other indicate that there is a need for more game content. We find this result predictable, as the current extent of the implementation is fairly limited in terms of content. However, we did get some interesting input on how to improve the game, such as for example introducing more objectives and a radar system to enhance the overview of the battlefield.

The pace of the game itself is commented on by some of the participants in question 9 ("What did you like about the game?"):

[1]This result is separated equally between the two groups of testers, with 2 and 6 for each respective result.

- *"Non-stop action"*
- *"Fast paced. Fun."*
- *"The theme. Gameplay and pace"*

Coupled with the fact that the comments on dislikes contain no references to the pace of the game this gives a clear impression of a preference in these participants of fast-paced gameplay in the style of the game prototype.

The question about prioritizing annoying game elements will be left out of the results. 7 out of 16 participants answered it incorrectly, indicating that the question was not stated clearly enough. As examples of incorrect answers on prioritizing using the numbers 1-5 is "1, 3, 3, 3, 4" or simply marking with crosses. A lesson learned is that not everyone know what "prioritize" means.

A Usability problem was identified. 6 participants disliked the handling of the movable turrets. As an example, here are two examples of answers to what they disliked about the game:

- *"The turrets. It was a pain to place them. Succeeded once!"*
- *"You have to be careful when placing the turrets, they fall over very easy"*

It is notable that while the turrets were problematic to use in the game, the concept of the turrets itself appealed to the participants, as visible in the following two quotes:

- *"Good survival game. Interesting idea with the turrets. Easy to get to know and play"*
- *"Lots of bug killing, and the strategic placement of turrets"*

The general opinion on the AI of the bugs was that they were avoiding dangerous areas by circumventing turrets and the player. A few participants felt that the bugs acted a bit too randomly, but the general consensus was that they seemed somewhat intelligent as apparent in the following quotes:

- *"They seemed to be fairly intelligent and tried to attack unguarded zones."*
- *"Good at going around the turrets."*
- *"The ants seemed to stay away from places under heavy fire, this makes it hard to keep track of where the ants are coming from."*

8.3. Further Development

The results of the Playtesting evaluation show that the basic concept of Smart Bugs works. This means that there is no reason to change this for the next iteration. Instead, the next iteration will focus on expanding the game content selection:

New levels and scenarios will be created, with a more thoroughly built story-line describing how the lone soldier has ended up deep inside enemy territory, and why he is not receiving any help other than turrets. The storyline should be un-intrusively presented in the game, in keeping with the results of the survey which describes the high

pace gameplay as a positive feature. As such, it should be presented through the messages shown at the beginning of each round, rather than through cinematics or other gameplay halting techniques.

The automatic turrets should be reworked to be easier to place and comfortable for the player to handle. Furthermore, some participants missed some common elements of FPS games, such as grenades and multiple weapons. In line with the games inspiration from the movie Starship Troopers, a grenade and a nuke launcher are possibilities. With the addition of more weapons, more types of bugs which are resistant to some weapons or react differently to danger levels are also a possibility.

The method developed for representing and using danger levels on pathfinding nodes seems suitable for this type of game. Although it does take a significant amount of tweaking to get right. An extension could be to give level designers more control of the parameters used by the algorithm, an example could be allowing the level designer to change the additional movement cost of danger levels within a specific area.

CHAPTER
9

Conclusion

This chapter will answer the two problem statements presented in section 1.1 on page 7.

Question 1: How can a bug learn to avoid dangerous areas in a map?

We started by exploring a number of AI techniques in chapter 4 and found out if there were existing implementations of these in the Source engine and how their implementation worked. Based on this we designed and described an AI system in section 6.2.

The basic idea of the system is to register danger in a pathfinding graph, which is used in an A* algorithm. When agents get killed in the game world, the danger level on nearby pathfinding nodes is raised. The updated danger level is reflected in the computation of a new path. This effectively means that the A* algorithm will calculate a different path for the next agents. To make agents retry previously avoided paths, four different algorithms for lowering the danger over time are suggested. The algorithms can be seen in section 6.2.1.2 on page 35.

Non-trivial parts of the implementation are detailed in chapter 7. The four above mentioned algorithms are compared in section 8.1 and algorithm IV is identified as the most suitable algorithm.

In section 8.1.2 on page 64 the entire AI implementation is tested and is found to be working as intended: The bugs will avoid dangerous paths where many bugs have been killed before, but retry a dangerous path if enough time passes without any bugs being killed there.

Question 2: How to investigate the impact such a bug has on User Experience?

To answer this question we started by exploring the concept of User Experience in section 5.1 on page 27. It turned out to be a very wide and not very well-defined field, so to get a more applicable understanding we delved deeper into existing game design guidelines in section 5.2 on page 28.

Our main discovery about the User Experience concept is that it is very hard to get a design right the first attempt. So a large part of User Experience is to evaluate a game during the development process and get input on how to improve it. Methods for doing evaluation were described in section 5.3 on page 30.

To try it in practice we conducted a Playtest evaluation with sixteen participants. The evaluated game was fully working but had a limited amount of game content. In addition to the knowledge and experience gained on using the method, we found that our game concept was fairly well received. We also discovered a number of issues and subjects for further improvement.

Generally speaking, our main lesson learned is that creating a good User Experience in a game is hard. The solution seems to be iterative development and continuous user participation. This point of view also fits well with widespread approaches to software development employed in the industry.

[1] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. Real-Time Rendering 3rd Edition. A. K. Peters, Ltd., Natick, MA, USA, 2008. 11, 13, 14

[2] John P. Davis, Keith Steury, and Randy Pagulayan. A survey method for assessing perceptions of a game: The consumer playtest in game design. In The International Journal of Computer Game Research. Microsoft Research, 2005. http://www.gamestudies.org/0501/davis_steury_pagulayan/. 30, 31, 74

[3] Melissa A. Federoff. Heuristics and usability guidelines for the creation and evaluation of fun in video games. Master's thesis, Department of Telecommunications of Indiana University, 2002. 28

[4] Olav Geil. The mathematical foundation of a*, 2006. http://www.math.aau.dk/~olav/undervisning/sp07/astarcor.pdf. 18

[5] Finn V. Jensen and Thomas D. Nielsen. Bayesian Networks and Decision Graphs. Springer Verlag, 2007. 25

[6] Effie Law, Virpi Roto, Arnold P.O.S. Vermeeren, Joke Kort, and Marc Hassenzahl. Towards a shared definition of user experience. In CHI '08: CHI '08 extended abstracts on Human factors in computing systems, pages 2395–2398, New York, NY, USA, 2008. ACM. 27

[7] Niamh McNamara and Jurek Kirakowski. Functionality, usability, and user experience: Three areas of concern. Interactions, 13(6):26–28, 2006. 27

[8] Ian Millington. Artificial Intelligence for Games. Morgan Kaufmann Publishers, 2006. 17, 19, 24, 26

[9] Jakob Nielsen. Usability Engineering. Morgan Kaufmann Publishers, 1994. 28, 30

[10] Usability.gov. Focus groups. http://www.usability.gov/methods/focusgroup.html. 30

[11] Wikipedia. Learnability. http://en.wikipedia.org/w/index.php?title=Learnability\&oldid=261339711. 28

[12] Wikipedia. Playability. http://en.wikipedia.org/w/index.php?title=Playability\&oldid=271076580. 28

[13] Wikipedia. Replayability. http://en.wikipedia.org/w/index.php?title=Replay_value\&oldid=276764549. 28

[14] Wikipedia. Software release life cycle. http://en.wikipedia.org/w/index.php?title=Software_release_life_cycle\&oldid=2834030769. 30

Part I

Appendix

APPENDIX

A

Test Handout

The hand-out used in the Playtest can be seen on the following five pages.

Welcome to the test of our game prototype.

The purpose of the test is to provide us with some feedback on the game. The overall goal is to test the game, not you.

The test procedure is as follows:

1) Answer a few questions on page 2.
2) Read the game instructions on page 3.
3) Play the game.
4) Answer the remaining questions on page 4 & 5

The total expected duration of the test is estimated to around 20-30 minutes.

The test is anonymous. We will use the results in our project report. As we may end up quoting parts of your comments directly, please write in English.

In advance thank you for your time, Group sp202a.

Please continue to the next page.

1) Please list your three all-time favourite computer games:

_____

_____

_____

2) Please list your three all-time favourite computer game genres:

_____

_____

_____

3) How would you characterize your previous experience with First-Person Shooter types of computer games? Mark your answer with X.

- ❑ Novice
- ❑ Intermediate
- ❑ Advanced

Please continue to the next page.

Please read the following instructions before playing the game:

The Objective:
Defend the main area (you will know it when you see it) against incoming bugs. The bugs win by getting into the main area and burrowing in. You goal is to kill the waves of incoming bugs. There are 11 waves of bugs to defend against, in order to win the game.

Game Controls:
The controls is standard for First-Person Shooter types of games. Use for example w,a,s,d keys to move around and aim with the mouse. The controls can be configured in the options menu.

An uncommon concept is the movable turrets. They automatically fire at enemies in their facing direction. To pick up a turret, stand next to it and press the use button (default: 'e'). Then move it by moving the player normally, and set it down again by pressing the use key once more. A turret can always be stood up, if it falls over, by picking it up.

Game Duration:
Give it a chance to learn the game. It might be hard at first. Feel free to restart the game if you do not succeed. Continue until you win the game, it is not a lengthy game. Otherwise, you are welcome to stop if you give up or do not feel sufficiently entertained any more.

Now it is time to play the game.

Do not flip to the next page before you have played the game. If you have played it, please continue and answer the remaining two pages of questions.

4) What is your overall impression of the game? Mark the most suitable answer with X.

- ❑ Overall positive
- ❑ Fairly positive
- ❑ Fairly negative
- ❑ Overall negative

5) Please prioritize the following game play elements in order of how much they annoyed you. Prioritize using the numbers from 1 to 5, where 5 is most annoying.

___ There were too many bugs.

___ There were too few weapons.

___ There were no power-ups

___ The gameplay was too monotonous

___ The turrets fell over too easily

6) How would you rate the difficulty in the beginning of a game? Mark the most suitable answer with X.

- ❑ Too easy
- ❑ Fair
- ❑ Too hard

7) How would you rate the difficulty towards the end of a game? Mark the most suitable answer with X.

- ❑ Too easy
- ❑ Fair
- ❑ Too hard

8) What is your impression of the behaviour and intelligence of the bugs?

_____

_____

_____

_____

9) What did you like about the game?

_____

_____

_____

10) What did you dislike about the game?

_____

_____

_____

11) What should we work on, in order to improve the game?

_____

_____

_____

_____

_____

That was all, thank you for your time.

Evaluation Results

Here is an extract of the results from the hand-outs. Only the most relevant answers is presented. There was 16 participants in the evaluation. "*Many Bugs*" and "*Hard Bugs*" denotes the two variations of the game.

The results of questions 1, 2, 5, 8 and 11 is omitted.

Question 3: How would you characterize your previous experience with First-Person Shooter types of computer games?

(**3**) Novice, (**3**) Intermediate, (**10**) Advanced

Question 4: What is your overall impression of the game?

Total:
(**4**) Overall Positive, (**12**) Fairly Positive, (**0**) Fairly Negative, (**0**) Overall Negative
"Many Bugs":
(**2**) Overall Positive, (**6**) Fairly Positive, (**0**) Fairly Negative, (**0**) Overall Negative
"Hard Bugs":
(**2**) Overall Positive, (**6**) Fairly Positive, (**0**) Fairly Negative, (**0**) Overall Negative

Question 6: How would you rate the difficulty in the beginning of a game?

Total: (**1**) Too Easy, (**14**) Fair, (**1**) Too Hard
"Many Bugs": (**0**) Too Easy, (**8**) Fair, (**0**) Too Hard
"Hard Bugs": (**1**) Too Easy, (**6**) Fair, (**1**) Too Hard

Question 7: How would you rate the difficulty towards the end of a game?

Total: (**0**) Too Easy, (**11**) Fair, (**5**) Too Hard
"Many Bugs": (**0**) Too Easy, (**4**) Fair, (**4**) Too Hard
"Hard Bugs": (**0**) Too Easy, (**7**) Fair, (**1**) Too Hard

Question 9: What did you like about the game?

"**Many Bugs**":
"Action and that it seems fairly simple"
"Good survival game. Interesting idea with the turrets. Easy to get to know and play"
"The turret idea"
"Non-stop action"
"Bugs! The concept, but need more content"
"Fast paced. Fun."
"It is a good concept and there is plenty of scope, for variation, and I enjoyed playing it."
"Lots of bug killing, and the strategic placement of turrets"

"**Hard Bugs**":
"Good difficulty. Fun to follow the bugs movements. Turrets shot red bugs, makes it

nicely difficult."
"I like the idea behind the game play. It mimics my favourite level from Half-life 2"
"The challenge. It was not easy."
"Tactical element with the turret immune bugs"
"The 'feel' of the game. Controlling the player was rather convincing combined with sounds. I really don't like bugs after this."
"The theme. Gameplay and pace"
"That i did not manage to win"
"Gameplay. Killing of bugs."

Question 10: What did you dislike about the game?

"**Many Bugs**":
"Too few weapons and no bigger bugs to kill"
"Found the weapon a little too inaccurate. I would prefer a slower weapon with higher accuracy"
"You have to be careful when placing the turrets, they fall over very easy. No way to keep track of the red bugs"
"Fairly quick felt monotone. After I failed at level 10 I didn't feel like starting over"
"The mechanics the standing the turret back up, seemed buggy and inconsistent"
"Kinda monotone. Mostly just the amounts of bugs that changes"
"The turrets. It was a pain to place them. Succeeded once!"
"Only the mounting of turrets"

"**Hard Bugs**":
"Corpses are distracting, however makes the game a bit harder"
"The weapons inaccuracy"
"The turrets were hard to place so they would be standing and not tipped over as soon as you stopped pressed 'e'"
"The turret mechanic is interesting, but very closely placing and replacing is difficult"
"Too monotonous, primarily"
"Turrets did not help enough. A lot to cover, one person"
"Not much new happens"
"Turrets fell over. Lag late in a game when many bug corpses were present"