A Voxel-Based Platform for Game Development

Thor Helms

Kongens Lyngby 2013
M.Sc.-2013-107

DTU Compute
Technical University of Denmark
Matematiktorvet, building 303B, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@compute.dtu.dk
www.compute.dtu.dk M.Sc.-2013-107

Summary (English)

The goal of the thesis is to create a game development platform for voxel based games, capable of creating games with a first person view. To this end, a conceptual game model has been created, which is used to define a game. A simple language has been created to define games consisting of a landscape, some game objects, and behavior for some game objects, along with a scanner/parser for this language, which outputs a conceptual game model. To transform the voxel models into graphics and display them in the game development platform *Unity*, a small voxel library has been created, which can output polygon models for a voxel model. Infinite landscapes are created using Simplex/Perlin noise. Behaviors are implemented via behavior trees (for games), but is still lacking an interpreter.

Summary (Danish)

Målet for denne afhandling er at lave en platform til udvikling af spil baseret på voxel grafik, der er i stand til at lave spil med et første persons billede af spillet. For at gøre dette er der lavet en konceptuel model af denne type spil, hvilket kan bruges til at definere et spil. Der er blevet lavet et simpelt programmeringssprog til at definere et spil i form af et landskab, nogle spil genstande og en adfærd til visse spil genstande, samt en skanner/oversætter til sproget, der giver en konceptuel spil model. For at lave en grafisk repræsentation af en voxel model er der blevet lavet et lille voxel bibliotek, som udregner en polygon-baseret model fra voxel data, og kan vise det i spiludviklingsværktøjet *Unity*. Uendeligt store landskaber kan laves via Simplex/Perlin støj. Adfærd er implementeret via adfærds træer (behavior trees) til spil, men i dette projekt er der ikke lavet en fortolker til disse.

Preface

This thesis was prepared at the department of DTU Compute at the Technical University of Denmark in fulfillment of the requirements for acquiring an M.Sc. in Informatics, and accounts for 35 of the 120 ECTS needed.

The counsellor for the project is Michael Reichhardt Hansen, with some initial support from Phan Anh Dung.

The student for this project is Thor Helms, student number s061377 at DTU.

Lyngby, 11-September-2013

Thor Helms

Acknowledgements

I would like to thank my counsellor Michael Reichhardt Hansen for the extensive support and guidance he has provided during my thesis, and Phan Anh Dung for the support he provided during the initial process of the project. Furthermore I would like to thank Rabie Jradi for being my sparring partner during the project, and for helping me put the finishing touches on the report.

On a personal level I would like to greatly thank my girlfriend Freja for her tremendous amount of patience and support. I would also like to thank my roommates Lauge and Sanne for having patience with me.

Contents

Chapter 1

Introduction

The game development process is a complicated process, which usually require large amounts of both creativity and technical skills. This makes it very difficult, if not impossible, for many people to realize their ideas for computer games, even if they have the creativity required to make amazing games. However, voxel based games, in particular Minecraft [3], allows anyone to let their creativity loose in 3D by having a construction-metaphor resembling that of real life, similar to building with Lego bricks. But as with Lego bricks, once a voxel construction has been completed, its purpose becomes merely an object of admiration, and its lifetime will likely come to an end. This is the motivation behind this project.

1.1 Goal

The goal of this project is to create a system that can be used to create voxel based games, with a particular focus on first person games with a single player. In both the game development platform and the games created with it, it should be possible to modify the world in detail.

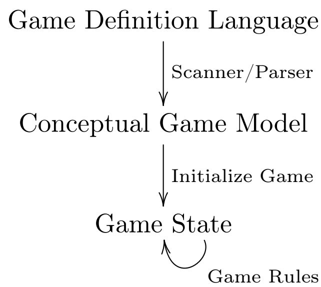


Figure 1.1: Overview of the system architecture of the game development platform.

1.2 Scope

Making a game development platform is a huge task, and thus not every aspect can be covered in a project of this size. Some focus will be given to a theoretical model of games and analyzing what it takes to create the different parts of a theoretical game. Regarding visual effects, the focus here is on transforming voxel models to polygon models, and letting another engine display the polygon models, with no focus on aspects such as animations, physical calculations and sound effects. Regarding the rules of a game, the focus is kept on how to define them using known techniques, and not how to implement these techniques.

1.3 Structure of the Thesis

The process for developing games followed in this project is outlined in figure 1.1. The first process is to create a game definition in a language invented here for the purpose, namely the *game definition language*. This is the only step that the game developer has to follow. The game definition is then transformed to an internal representation, the conceptual model, via a scanner and parser. Finally, in order to actually play the game, a formal game and the initial state are created, which is iterated until the game has ended.

Chapter 2 gives an introduction of the basic concepts used in this project. Chapter 3 gives a brief overview of the structure of the solution, to guide the understanding when reading the report. Chapter 4 describes a voxel library that has been created to visualize voxel models in Unity. Chapter 5 defines the conceptual model. Chapter 6 gives some examples of the created game definition language as well as defines the complete parser for the language. Chapter 7 partially describes the process of converting from the conceptual model into an actual game, and defines the types used during execution of a game. Finally, chapter 8 gives a brief overview of how to test a project as this, chapter 9 outlines the results of the project, and chapter 10 discusses the project and gives some recommendations as to future work of the project.

1.4 Notation

There are three programming languages used in this report. The first is that of the game definition language. This is highlighted as seen below.

```
Game Definition Language looks like this
```

The second language is the definition of the parser, which is used as input to FSYacc to create a parser in F#. The parser definitions are highlighted as seen below.

```
Parser−definition looks like this
```

The final language used is F#, which is also used as the pseudo code in this report. This is highlighted as seen below.

```
F# code looks like this
```

Throughout the project, the commercial game development platform Unity has been used as a platform.

1.5 Related Work

The main inspiration of this project is the game Minecraft, which itself is inspired by the game Infiniminer [4]. These two games have a data structure and visual appearance based on voxels, displayed as cubes. There are other ways of displaying voxels, for instance by using the Marching Cubes [5] or the Transvoxel^{TM}[6][7] algorithms, which both make a more smooth surface than cubes.

An implementation of the Simplex/Perlin noise algorithms [8] [9] is used in this project, the source of which can be found in appendix B.4. Simplex/Perlin noise has found application in many games, Minecraft included.

The concept of *behavior trees* [10] has already been used in games, for instance in Spore [11] [12]. The term *behavior trees* is also used in classical software engineering, but bears little resemblance with the behavior trees used in games. The theory used in this project is explained in more detail in section 2.2.6.

Attempts to establish what a game exactly is has been done most recently by Jesper Juul [1] and Katie Salen & Eric Zimmerman [2], and multiple others before them. The definition of a game used in this project is given by Salen & Zimmerman, as they deal with a more formal view of what a game is, contrary to defining games from abstract terms such as *fun* and *player effort*.

As mentioned, this project makes use of the game development platform Unity [13]. This is of course far from the only game development platform on the market.

Chapter 2

Requirement Specification

In this chapter follows a purposely vague definition of a game dubbed *Carl of Sheeponia*; a game brought to existence for demonstration purposes. This is followed by an analysis of the basic concepts in that type of game, followed at last by a requirement specification for a game development platform for that type of game.

2.1 Case Study: Carl of Sheeponia

The mountainous region of Sheeponia is the home of tremendous amount of sheep. And as always, with sheep come the wolf, preying on the lonely sheep or weak individuals from sheep herds.

Sheep tend to stay in large herds of sheep, and are cautious when humans are present. Their main food is grass, which is abundant in the valleys of Sheeponia. They fear wolves, except when the amount of sheep greatly outnumber the amount of wolves, in which case the sheep will return aggressive attitudes from the wolves. In all other cases, the sheep will flee when faced with aggression from one or more wolves.

The wolves of Sheeponia can be fearsome creatures, that may stalk its prey for long periods, until it feels confident it can kill it. When faced with an aggressive opponent, wolves will return the aggression. They prey on lonely or weak sheep, and on occasion humans. Wolfs tend to hunt when in packs, but is also often seen alone.

Our hero, Carl, is a human, who finds him self in the wilderness of Sheeponia, armed with a rifle and a sword. The goal of Carl is simply to survive.

2.2 Basic Concepts

Based on the above example of a game, this section will introduce some concepts used throughout the report, which are used to describe an abstract game.

The virtual world that the game takes place in, e.g. Sheeponia, can be split into two categories: The *landscape*, and *game objects*, with the landscape being the mountainous regions etc. of Sheeponia, i.e. a large area which is largely static, and game objects being Carl, his rifle and his sword, the sheep, and wolfs.

2.2.1 Game Objects

Game objects can be split into two categories: Inanimate objects (*items*), and creatures, where the difference is whether or not they can interact with the world by their own initiative, i.e. they have a *behavior*. Player characters can be viewed as creatures, and thus creatures can be split into two categories: *Player characters* and *non-player characters* (NPCs).

Items As mentioned above, items are inanimate, and can thus be identified merely by their visual characteristics, or *appearance*, and some metaphysical *attributes*. The items present in the Carl of Sheeponia definition given in section 2.1 are the rifle and sword that Carl possesses.

Non Player Character (NPC) NPCs are identified as items, i.e. with a visual appearance and some metaphysical attributes, and additionally with a behavior.

Unlike that of a player character, the behavior of an NPC is completely autonomous, acting according to its surroundings.

Player Character While player characters are similar to NPCs, they are also slightly more advanced. In Carl of Sheeponia, Carl has two weapons, which he must be able to store somewhere. For this reason, player characters have an *inventory* of items, besides the characteristics of an NPC.

Furthermore, because a player character is controlled by a human, who must be able to see the game from some perspective, a player character also have a camera-definition, which defines from which perspective the human player sees the game.

The behavior of player characters is what defines how the player character should react to the input given by the human player.

2.2.2 Landscape

It is assumed that games will only have one landscape. The landscape is identified by being relatively static and volumetrically the majority of game content, whereas objects are far more dynamic and can be interacted with by the player. While a game as Minecraft allows the player to change the landscape by adding and removing cubic voxels, this is still restrained within the grid structure of the voxel data structure, whereas for instance a rifle can take up any location and have any rotation within cavities in the world.

Creating a landscape by hand can be a tedious task due to the size of the landscape, and a repetitious landscape is usually not desired in games. Some games have procedurally generated landscapes, often using Perlin or Simplex noise, which can both create a randomized multidimensional noise that can mimic natural structures. The use of procedurally generated landscapes can greatly reduce the complexity of creating large landscapes that still seem natural. An example of a voxel based landscape created in this manner, from the game Minecraft, can be seen in figure 2.1.

When defining a procedural landscape, certain operations should be possible. Below I give a list some attributes, along with a semantic for them.

Height map A height map defines a y-value at each (x, z) coordinate, and creates a surface which can be used to distinguish between two different landscapes: One below the surface, and one above it.

Area map An area map uses a height map to distinguish between different (x, z) areas. For instance, if a flat map of the earth was used as a height map, an area map could define the different countries. An area map can contain multiple y-ranges, each containing their own landscape.

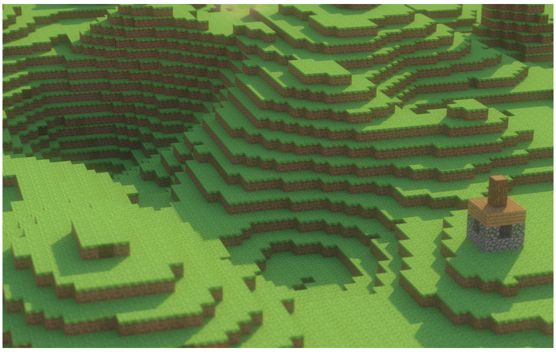


Figure 2.1: Procedurally generated landscape from the game Minecraft. Source in appendix B.3.

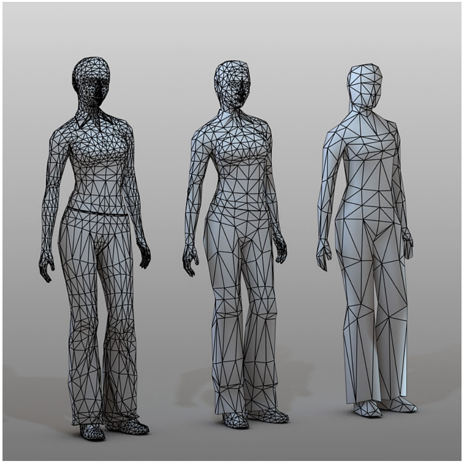
Volume map This is similar to a height map or an area map, except that it is 3-dimensional noise. This means that each (x, y, z)-coordinate has a numerical value in some range. Like an area map, a volume map may contain multiple landscapes defined within a certain range of the previously mentioned numerical value.

2.2.3 Visual Objects

Game objects and the landscape also need to be visualized as 3D objects. The most commonly used representation for 3D objects in games is that of polygon based objects, an example of which can be seen in figure 2.2a. These consist of one or more polygon meshes and typically one or more *textures*, which are 2D images laid on top of the polygons. Polygon based objects can provide for realistic looking objects even with a relatively low *polygon count*.

Another way of representing 3D objects is that of voxel based graphics, which in recent years has been made vastly popular through the game Minecraft [3]. In it, everything consists of voxels rendered as cubes, similar to that of figure 2.2b. However, the cubes are rendered as polygons, as a contrast to the much more computationally heavy ray-casting method [14], which is infeasible on todays polygon-optimized hardware.

Besides from creating cubic polygon models from voxels, more smoothed polygon models can also be created via the Marching Cubes [5] and Transvoxel^{TM}[6][7] algorithms. However, both techniques make it more difficult to visually distinguish where one voxel ends and another begins.



(a) Polygon based character.

(b) Voxel based character.

Figure 2.2: Two different types of 3D graphics. Source in appendix B.1 and B.2 respectively.

A difference between polygon based representation and voxel based representation of 3D models is that polygons only describe the surface of an object, whereas voxels describe the entire volume. This means that manipulation of voxel models require less complex algorithms, and that many physical properties such as weight and deformation can be simulated with ease. Furthermore, manually changing a voxel based object or landscape has proven very easy to do for an untrained user.

2.2.4 What is a game?

The following definition of a game, by Salen and Zimmerman [2], can be used to determine what a game is and consists of: "*A game is a system in which players engage in an artificial conflict, defined by rules, that results in a quantifiable outcome.*" Thus we can see that a game consists of four parts: Some *players*, an *artificial conflict*, some *rules* and an *outcome*.

The artificial conflict takes place over time, and at any single point in time, the artificial conflict is in some *state*, which depends on the state of the various game objects at that point in time.

The rules of a game is possibly the most important aspect of a game, that a game developer should to be able to change. In a game as Carl of Sheeponia, the behavior of the sheep, wolfs and Carl are what the game developer should be able to change. A technique that can be used to define behavior is that of *behavior trees*. These have successfully been used in several games.

Other aspects of game rules may be assumed given, such as physical calculations and rendering of the graphics.

2.2.5 Levels in a Game

While there isn't in Carl of Sheeponia, other games may have a notion of different *levels* within that game. An extension to Carl of Sheeponia where the notion of levels might be needed, would be if Carl could travel to the moon. Then the area of Sheeponia would be one level, and the area of the moon would be another.

As with game objects, a game may have some metaphysical attributes. These should be manifested in the levels, as they may change from level to level.

An assumption here is that all levels in a game share the same game object definitions, behaviors, rules and landscape definitions, and the distinction between the different levels is what landscape and game objects they actually use, along with the metaphysical attributes.

2.2.6 Behavior Trees

The behavior of game objects, including the player character, is in this project created via *behavior trees*, a technique already used successfully in multiple games.

Behavior trees are in their simplest form trees. It contains a number of different types of nodes. One of the reasons to use behavior trees is that their graphical nature (they are typically edited in a graphical editor) allows for designers, in contrast to programmers, to define the behavior of the NPCs.

A behavior tree, and thus the different nodes, can have one of three return states when evaluated: *Success*, *fail* and *running*. The semantics of the *success* and *fail* states are similar to a boolean value, while the semantic of the *running* state is that it is yet to complete and should thus be queried again. For example, if a sheep should walk for 10 seconds, then until the 10 seconds has passed, the *walk* action will return with a *running* state.

What type of nodes a behavior tree consists of differs a bit from source to source, probably because it is a tool and not a theory, but below are some common node types and their semantics.

Sequence Semantically similar to an *and*-list, in that it contains multiple child nodes and will evaluate them in a serial manner until one fails, in which case the *sequence* node fails. If all child nodes succeed, the *sequence* will likewise succeed. If a child node returns with the *running* state, the *sequence* node will also return the *running* state, and the next time the *sequence* node is queried it will continue with the child node that returned with the *running* state.

Selector Similar to the *sequence* node, but negated, and thus semantically similar to an *or*-list. Evaluates child nodes in a serial manner until one succeeds, in which case the *selector* node succeeds. Like the *sequence* node, at most one child node can be in the *running* state at a time.

Parallel The *parallel* node contains multiple child nodes. There is no definition of the order or the manner, as long as all child nodes are evaluated even if one is in the *running* state. This means that the evaluation can be serial as with the *sequence* and *selector* nodes. Unlike the *sequence* and *selector* nodes, any number of child nodes of the *parallel* node can be in the *running* state at a time.

The semantics for when a *parallel* node succeeds or fails is a little unclear, so I apply the semantics that the *parallel* node either succeeds or fails if some pre-specified number of child nodes succeeds or fails, and until this happens it keeps running and evaluates its child nodes.

Decorator This is not actually a specific node, but a class of nodes. *Decorator* nodes have exactly one child node, and are used to change the behavior of the subtree that is its child node. *Decorator* nodes can for instance change the returned state of its child or restrict how often its child node is evaluated. There is no specific set of *decorator* nodes to be included in an implementation, and no matter the set it should optimally be possible for the game developer to extent the set of *decorator* nodes.

Link Reference to another behavior tree. This ensures that it is possible to make modular trees, where multiple different behaviors share sub-behaviors. Upon evaluation, the *link* node will return the same value as the referenced behavior tree.

Condition Contains a boolean expression. If the boolean expression evaluates to true, the *condition* node returns successfully, and if it fails, so does the *condition* node.

Action The *action* node evaluates a function-call, which must return with the same state as a behavior tree, i.e. *success*, *fail* or *running*, which is also the returned state of the *action* node.

Note that I earlier stated that behavior trees are trees. I will now reveal that this is slightly untrue, given the nature of the *link* node. While it is true that a behavior tree is a tree when ignoring the link node, the link node actually transforms it into a directed and possibly cyclic graph. In any implementation, care must be taken to ensure that the behavior trees are acyclic directed graphs.

The semantics for the different return states are clear when dealing with nodes in a tree, but they are not completely clear when speaking of the return state for an entire tree. Therefore I have chosen to define a semantic that relates to the *quantifiable outcome* of a game, namely that if a behavior tree for a player returns with the state *success*, then the game has been won for that player, and if it returns with the state *fail*, then the game has been lost. For an NPC, the semantics is that if its behavior tree returns in a different state than *running*, then that NPC will be removed from the game.

While the *action* and *condition* nodes are similar to some extent, an additional semantical difference between them can be defined, lending an observation from functional programming: *Conditions* may not have side effects, while *actions* may. This is an optional semantical addition to the behavior tree semantics.

In figure 2.3 can be seen an example of a partial behavior tree. The root is a parallel node, containing a node defining the winning condition and a node defining the losing condition, and a node containing some actions that has to be performed at all times.

2.3 Game Development Requirements

As the output of this project should be a platform for developing games, it should enable the user (game developer) of this product to have a great degree of freedom. With greater degree of freedom also comes lack of focus, so to retain the focus on developing games, the requirements will be based on a platform that can create games within the genre *First Person Shooter*, in which the view of the game is similar to the view of a person in real life, i.e. first person.

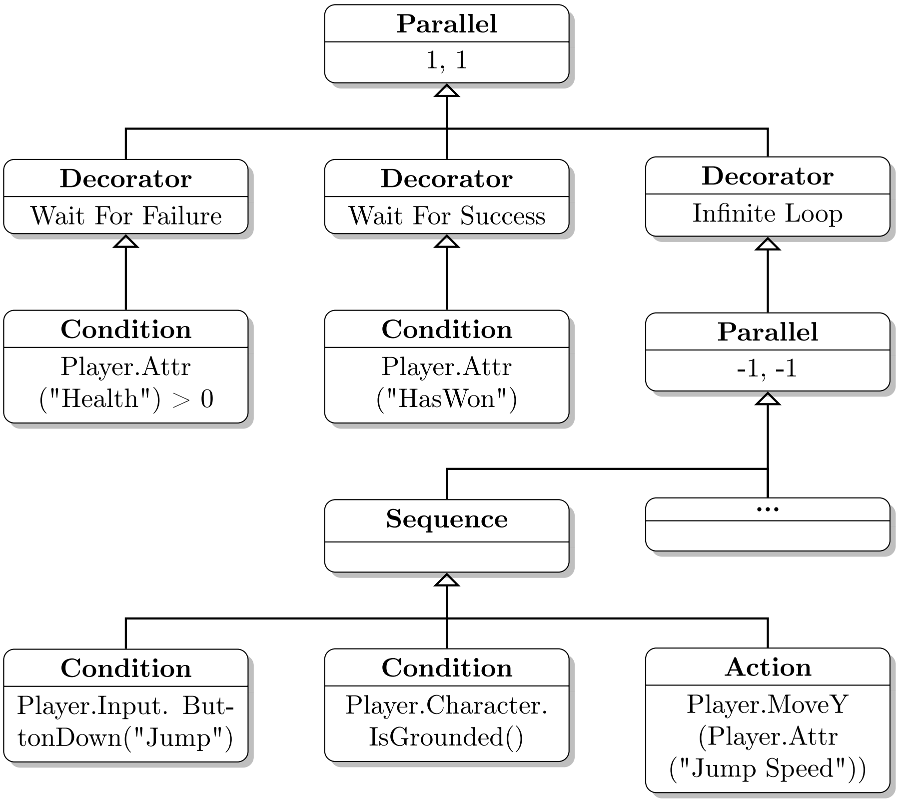


Figure 2.3: A partial sample of a behavior tree from Carl of Sheeponia.

1. The game should take place in a simulated 3-dimensional world.
2. Simulation of a physical environment, including gravity, should be taken as granted.
3. It should be possible to freely form a landscape in which the game takes place.
4. It should be possible for the player of a game to alter the landscape.
5. It should be possible to freely shape the objects and creatures of the game.
6. It should be possible to freely determine the behavior of the creatures in the game.
7. It should be possible to freely design and implement the game mechanics.
8. It should be possible to freely make the rules for when a game has ended.
9. It should be possible to freely create the initial state of the game.
10. Progression of the game, from one state to the next state, should happen automatically based on the rules, game mechanics and winning/losing conditions input by the game developer.

Chapter 3

System Architecture

Before presenting the different aspects of the implementation in this project, a system overview will be given in this short chapter.

The system is roughly divided into four libraries, each consisting of multiple modules. The most basic library is the voxel library, which describe a data structure for voxel graphics, and provides methods for rendering voxel graphics through Unity.

Next follows a conceptual game model, which defines abstract syntax in which a game can be defined. A scanner/parser combination can transform a script in a custom made language, into the model defined in the conceptual game model.

Finally there is a formal game model, which defines how a game behaves when executed.

An overview of the system and the relations between the different libraries can be seen in figure 3.1.

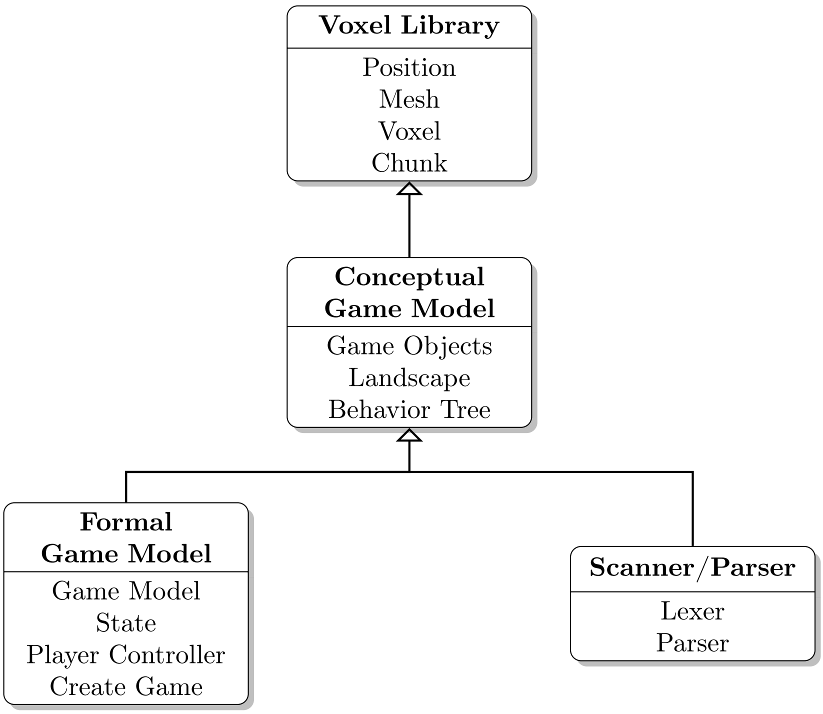


Figure 3.1: Overview of the different modules created in this project.

Chapter 4

Rendering Voxel Models

To avoid having to visualize each individual voxel, and avoid having to potentially reconstruct large areas of the world after minor changes, the voxel data structure is divided into *chunks*. A chunk is hyper-rectangular dense collection of voxels, implemented as a 3-dimensional array along with the offset of the chunk.

A landscape consists of a number of non-overlapping chunks of the same size, ordered in a hyper-rectangular 3-dimensional array, and the appearance of a game object can be described as a single chunk of some size. Doing this, and considering the different chunks of a landscape to be independent, allows for the visualization procedure to only have to consider single chunks.

This chapter describes an implemented voxel library, consisting of four modules.

4.1 Position

As the voxel library deals with 3-dimensional data, a *Position* type has been created, denoting a position in 3D space, which can be seen below.

```
type Position = float32 * float32 * float32
```

When dealing with axis aligned cubes, as voxel representations sometimes (and in this project) are, the six faces can be described as being either the positive or negative x-, y- or z-side. This is captured by the *Direction* type below.

```
type Direction =
    | XPos | XNeg | YPos | YNeg | ZPos | ZNeg
```

Some basic operations are supported on the *Position* and *Direction* types, for which the signatures can be seen below.

```
val PositionBinop:
    ('a -> 'b -> 'c) -> 'a * 'a * 'a ->
    'b * 'b * 'b -> 'c * 'c * 'c
val PositionUnop:
    ('a -> 'b) -> 'a * 'a * 'a -> 'b * 'b * 'b
val Distance: Position -> Position -> float32
val IncreasePositionInDirection1:
    Direction -> Position -> Position
val IncreasePositionInDirection2:
    Direction -> Position -> Position
```

The two functions *PositionBinop* and *PositionUnop* can be used to perform simple operations on one or two positions, such as addition or type conversion. The *Distance* function simply calculates the euclidian distance between two positions.

The two functions *IncreasePositionInDirection1* and *IncreasePositionInDirection2* will, for a given direction, increase a given position in one of the other two directions. As an example, the position $(x, y, z) = (a, b, c)$ will for the directions *XPos* and *XNeg* will return either $(a, b + 1, c)$ or $(a, b, c + 1)$.

The complete source code for the *Position* module can be seen in appendix C.3.4.

4.2 Mesh

As mentioned in section 2.2.3, voxels can be visualized as polygon models. This is the basis for visualization used in this project. To match the data structure used by Unity for visualization, the voxel models have to be transformed into a list of polygon meshes, each list containing only polygons of a single color, and each mesh must consist of a list of vertices and a list of triangles, where the list of triangles is a list of integers of length $3n$, where n is the number of triangles in the list. This is captured by the *Mesh* type below.

```
type 'a Mesh =
    (UnityEngine.Vector3 list) * (int list) * ('a)
```

The *Mesh* module contains a function to combine two lists of meshes, such that meshes with the same color in the two lists will be concatenated to a single list. The signature for this function can be seen below.

```
val CombineMeshes:
    ('a * 'a -> bool) -> 'a Mesh list ->
    'a Mesh list -> 'a Mesh list
```

The full source code for the *Mesh* module can be seen in appendix C.3.5.

4.3 Voxel

As the amount of voxels may sometimes be in the millions, a single voxel is implemented here with a 16 bit (unsigned) integer, as seen below. By using a primitive type, it is guaranteed that there is no overhead, as there may be for other types.

```
type Voxel = uint16
```

With this implementation, the position of a voxel is not explicitly stored with the representation for the voxel.

In the 16 bits used to represent a voxel, the last 12 bits represent the red, green and blue colors respectively, each represented with 4 bits. Only one of the first four bits are used, and it is used to mark a solid as solid, i.e. that it exists.

The *Voxel* module contains multiple functions to perform operations on/with voxels. The signature for the most important functions can be seen below.

```
val VoxelIsVolid: Voxel -> bool
val VoxelFromString: string -> Voxel
val VoxelToMesh:
    Direction -> Position -> Voxel -> Voxel Mesh
```

The function *VoxelIsSolid* looks at the relevant bit in a voxel representation to determine whether or not it is solid.

The function *VoxelFromString* expects a string matching the regular expression $\hat{}\#[0-9a-fA-F]\{3,3\}\$$, for example $\#F0F$ which denotes a voxel with maximum red and blue, and no green. Voxels created with this function will always be solid.

VoxelToMesh transforms a single voxel to a mesh for the face identified by the given direction, by creating the two triangles that face consists of.

The source code for the *Voxel* module can be seen in appendix C.3.6.

4.4 Chunk

As mentioned earlier, the data for a voxel model, be it a landscape or a game object, is divided into one or more chunks. The implementation of the chunk type can be seen below.

```
type ChunkData = Voxel [,,]
type Chunk = ChunkData * Position
```

The Chunk module contains multiple functions for calculating on/with chunks, the signature of the most important which can be seen below.

```
val ChunkdataFromString:
    int -> int -> int -> (char -> Voxel option) ->
    String -> ChunkData
val ChunkToMesh:
    Direction -> Chunk -> Voxel Mesh list
val DisplayMeshes:
    UnityEngine.GameObject -> Chunk ->
    UnityEngine.Material -> Unit
```

The function *ChunkdataFromString* creates a chunk of some given dimensions (the first three arguments), with the data from a string using a given function to convert a single character to a voxel. For example, converting 0 to a non-solid voxel and 1 to a solid voxel of some color, the string '01011111' could denote a $2 \times 2 \times 2$ chunk with 6 solid voxels.

ChunkToMesh converts a chunk to a list of meshes, creating only the meshes that face a given direction. In order to minimize the size of the created polygon meshes, a few techniques have been applied. First of all, neighboring voxels of the same color are created as larger triangles covering all the voxels in rectangles.

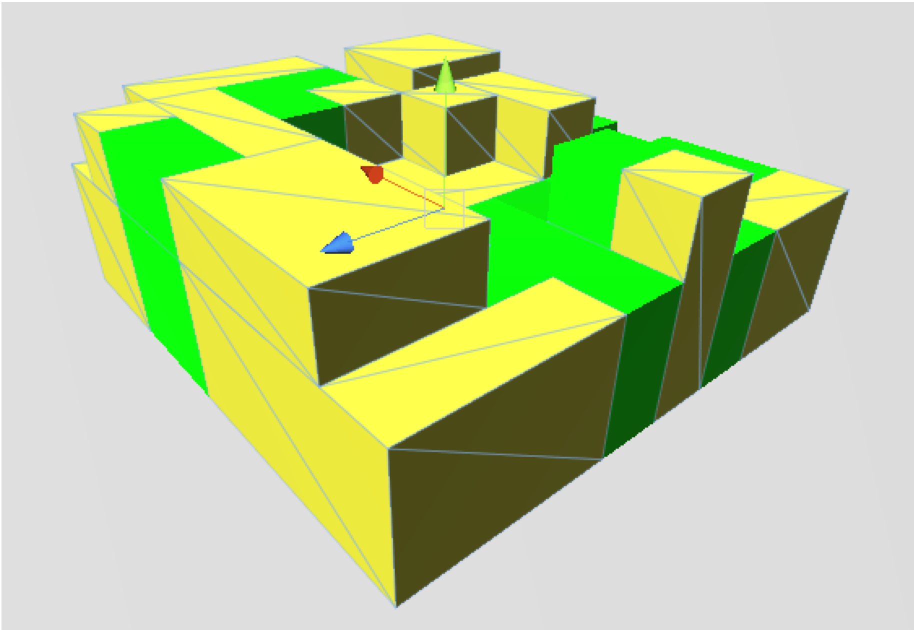


Figure 4.1: Example of a chunk converted to polygon meshes.

Secondly, voxels that can't be seen are not visualized. An example of a chunk mesh created in this way can be seen in figure 4.1, where it can be seen that the meshes span rectangles of similarly colored voxels.

ChunkToMesh is implemented as a greedy algorithm. It iterates over all positions in the chunk, and for each position it attempts to greedily create a mesh if the voxel at that position is visible and no mesh has been previously created for that position. For a position for which a mesh needs to be created, the mesh is maximized with a preference towards square or close to square meshes. A neighboring voxel can be included if it is a voxel of the same color for which no mesh has been created, or if it can't be seen because it is blocked by a solid voxel.

The function *DisplayMeshes* takes a chunk and creates the meshes for it in all six directions, and assigns the calculated meshes to a Unity game object (*GameObject*), allowing it to be rendered by Unity's engine.

The source code for the *Chunk* module can be seen in appendix C.3.8.

Chapter 5

Conceptual Game Model

Building on the basic concepts from section 2.2, this chapter gives a formal definition of these concepts.

5.1 Game Objects

As mentioned, all three types of game objects (items, NPCs and player characters) have an appearance and some attributes. Both of these can be thought of as being a list of variables. The value of variables are here formalized as being one of five primitive values: Integers, floats, strings, boolean values, or 3-dimensional positions. This can be formalized as such:

```
type PrimitiveValue =
    | PrimInt of int
    | PrimFloat of float32
    | PrimStr of string
    | PrimBool of bool
    | PrimPosition of float32 * float32 * float32
```

The source code for the game objects listed below can be seen in appendix C.3.10.

5.1.1 Items

Using the above definition of primitive values, items can be formalized as a mapping from a variable name to its value, for both its appearance and its attributes, as seen below.

```
type ItemDef =
    {
    Appearance : Map<string, PrimitiveValue>;
    Attr : Map<string, PrimitiveValue>;
    }
```

5.1.2 Non Player Characters

The definition of an NPC is very similar to that of an item, except that an NPC has a behavior, in this case in the form of a behavior tree. Assuming the behavior tree is defined elsewhere with a name as key, an NPC can be modeled as seen below.

```
type NpcDef =
    {
    Appearance : Map<string, PrimitiveValue>;
    Attr : Map<string, PrimitiveValue>;
    BehaviorTree : string;
    }
```

5.1.3 Player Characters

The player character is in many ways similar to an NPC, except that it has a description of how the point of view of the game should be for the human player, in the form of a *camera* description, as well as an inventory. Items in the inventory has a name, and assuming that the items are defined elsewhere and can be accessed via the item name, the inventory can be modeled as a map from the inventory-name to the item-name. Below is a formal definition of a player character.

```
type PlayerDef =
    {
    Appearance : Map<string, PrimitiveValue>;
    Attr : Map<string, PrimitiveValue>;
```

```
    BehaviorTree : string;
    Camera : Map<string, PrimitiveValue>;
    Inventory : Map<string, string>;
    }
```

5.2 Landscape

Defining a landscape can be accomplished in myriads of ways. Here I will focus on a procedural definition of potentially infinite landscapes, using 2D and 3D noise, which is formally defined in the following sections.

5.2.1 Height Maps

A height map is essentially a 2D noise map, using x- and z-coordinates to calculate a y-coordinate. Four basic types of a height map are laid out below.

```
type Heightmap =
    | Noise2D of float32 * float32 * float32
    | Plane of float32
    | Add2D of Heightmap list
    | Offset2D of Position * Heightmap
    | HMRef of string
```

The first type is a noise map with two parameters for the horizontal size of the noise, and a third parameter for the vertical height of the noise. The second type is a plane, which defines a height map of constant height in all (x, z) positions.

The third type of height map is a sum of multiple height maps. The fourth type is an offset of a height map, which offsets a height map by a 3-dimensional vector. The last type of height map is a reference to another height map, under the assumption that there is some environment containing height maps.

5.2.2 Volume Maps

Volume maps are in most senses similar to height maps, except that the noise takes an extra parameter, corresponding to the extra dimension in the noise.

Furthermore, a volume map can't have a plane, the same as a height map can. Below is a formal definition of a volume map.

```
type Volumemap =
    | Noise3D of float32 * float32 * float32 * float32
    | Add3D of Volumemap list
    | Offset3D of Position * Volumemap
    | VMRef of string
```

With the noise, the first three parameters define the sizes of the noise along each axis, while the fourth parameter defines the weight of that particular volume map.

5.2.3 Landscape Procedure

Besides the operations described in section 2.2.2, it should be possible to define a landscape consisting solely of game objects or voxels, and it should be possible give a reference to landscapes or voxels defined elsewhere. A formal description of a landscape can be seen below.

```
type LandscapeDef =
    | Heightmap of Heightmap * LandscapeDef * LandscapeDef
    | AreaMap of Heightmap * (Range list) * LandscapeDef
    | VolumeMap of Volumemap * (Range list) * LandscapeDef
    | LandscapeRef of string
    | Gameobject of string * string
    | VoxelVal of Voxel
    | VoxelRef of string
and Range = float32 * float32 * LandscapeDef
```

5.3 Levels

A level should have a name of the landscape in which the level takes place, as well as the position at which the player starts and some attributes. The attributes of a level have the same form as those of game objects. A formal description of a level can be seen below.

```
type LevelDef =
    {
    Landscape : string;
```

```
    PlayerSpawnPoint : Position;
    Attr : Map<string, PrimitiveValue>;
    }
```

5.4 Behavior Trees

Following the semantical description of a behavior tree in section 2.2.6, a formal model of a behavior tree can be built as seen below.

```
type BehaviorTree =
    | Sequence of BehaviorTree list
    | Selector of BehaviorTree list
    | Parallel of int * int * (BehaviorTree list)
    | Decorator of ActionExpr * BehaviorTree
    | Link of string
    | Condition of Expr
    | Action of ActionExpr
```

The behavior tree model must encompass, not just the behavior tree nodes described in section 2.2.6, but also the expressions inherent in the *decorator*, *condition* and *action* nodes as seen above. These are however left out of this project, the reason for which is discussed in section 10.1.

The source code for the behavior trees, and a preliminary definition of expressions, can be seen in appendix C.3.9.

5.5 Game Definition

Using the definitions from earlier in this chapter, a game definition can be seen below.

```
type GameDef =
    {
    Levels : Map<string, LevelDef>;
    Player : PlayerDef;
    Items : Map<string, ItemDef>;
    Npcs : Map<string, NpcDef>;
    BehaviorTrees : Map<string, BehaviorTree>;
    Voxels : Map<string, Voxel>;
```

```
    Heightmaps : Map<string, Heightmap>;
    Volumemaps : Map<string, Volumemap>;
    Landscapes : Map<string, Landscape>;
    }
```

Note that there is only one player, and all items, NPCs, levels, behavior trees and various aspects of the landscape have a unique name within each type. This means that there can both be an NPC with the name ‘Sheep’, as well as a behavior tree with the name ‘Sheep’.

The source for the game definition above, as well as the level definition (and game object definitions) can be seen in appendix C.3.10.

Chapter 6

Game Definition Language

In this chapter, I give some examples of a language that can be used to create the conceptual game model described in chapter 5, along with a parser for this language. The language is referred to as the *game definition language*.

The complete source for the parser can be seen in appendix C.2.2, and the source for a lexer for this language can be seen in appendix C.2.1. The examples of a game definition language used throughout this chapter is from a definition of the game Carl of Sheeponia, and can be seen in its entirety in appendix C.1.1.

6.1 Generic Parts of the Parser

The parser contains some generic parts, which are used by the other definitions. These generic are described without examples in this section.

A position, being a 3-dimensional coordinate, can be parsed via the following parse rule.

```
Position:
    LPARAN Num COMMA Num COMMA Num RPARAN
    { ($2,$4,$6) }
```

Descriptions, such as the appearance and attributes of an item, are given by a list of definitions. This is embodied in a *named list*, for which a parse rule can be seen below.

```
NamedList:
    | { [] }
    | STRING EQ PrimitiveValue NamedList
      { ($1,$3)::$4 }
```

The rule above uses the construction of primitive values. A parse rule for these can be seen below.

```
PrimitiveValue:
    | INT       { PrimInt $1 }
    | FLOAT     { PrimFloat $1 }
    | STRING    { PrimStr $1 }
    | TRUE      { PrimBool true }
    | FALSE     { PrimBool false }
    | Position  { PrimPosition $1 }
```

In some (but not all) cases, there should be no distinction between an integer and a float, which is embodied by the following parse rule.

```
Num:
    | INT   { $1 |> float32 }
    | FLOAT { $1 }
```

6.2 Items

An example of a definition of an item, namely a sword, can be seen below. It contains two parts, one describing the appearance, and one describing the metaphysical attributes.

```
Item Sword = (
    Appearance = (
        "Width" = 1
        "Height" = 8
        "Depth" = 3
        "Color" = "#B83"
        "Chunk" = "010 010 111 010 010 010 010 010"
        "Center" = (0.5, 1.5, 1.0)
        "Scale" = 0.2 )
```

```
    Attr = (
        "Weapon Type" = "Melee"
        "Attack Strength" = 15 ) )
```

Starting from the first set of parentheses, the item can be parsed via the following parse rule:

```
ItemDef:
    APPEARANCE EQ LPARAN NamedList RPARAN
    ATTR EQ LPARAN NamedList RPARAN
    { {
        Appearance = Map.ofList $4;
        Attr = Map.ofList $9;
    } }
```

6.3 Non Player Characters

As mentioned earlier, an NPC is similar to an item, except it has some behavior, in this case in the form of a behavior tree. Excluding the appearance and attribute definition, which has been seen in section 6.2, an NPC can be defined as seen below, where the behavior tree is defined as a pointer to an otherwise defined behavior tree.

```
NPC Sheep = (
    Appearance = ( ... )
    Attr = ( ... )
    BehaviorTree = SheepBehavior )
```

An NPC can be parsed with the following parse rule.

```
NpcDef:
    APPEARANCE EQ LPARAN NamedList RPARAN
    ATTR EQ LPARAN NamedList RPARAN
    BEHAVTREE EQ ID
    { {
        Appearance = Map.ofList $4;
        Attr = Map.ofList $9;
        BehaviorTree = $13;
    } }
```

6.4 Player Characters

The definition of a player character resembles that of an NPC, with the addition of a camera and inventory definition. The camera is defined in the same manner as the appearance and attributes, while the inventory resembles the behavior tree definition in list form. An example of a player definition can be seen below.

```
Player = (
    Appearance = ( ... )
    Attr = ( ... )
    BehaviorTree = PlayerBehavior
    Camera = (
        "Center" = (0,-1,0)
        "Facing" = "Forward" )
    Inventory = (
        "Weapon" = MeleeWeapon ) )
```

The player definition above can be parsed via the following parse rule.

```
PlayerDef:
    APPEARANCE EQ LPARAN NamedList RPARAN
    ATTR EQ LPARAN NamedList RPARAN
    BEHAVTREE EQ ID
    CAMERA EQ LPARAN NamedList RPARAN
    INVENTORY EQ LPARAN ItemList RPARAN
    { {
        Appearance = Map.ofList $4;
        Attr = Map.ofList $9;
        BehaviorTree = $13;
        Camera = Map.ofList $17;
        Inventory = Map.ofList $22;
    } }
```

The player's inventory is defined as a list of item references. A parse rule for this can be seen below.

```
ItemList:
    | { [] }
    | STRING EQ ID ItemList
      { ($1,$3)::$4 }
```

6.5 Height Maps

An example of a height map in the game definition language can be seen below. It consists of four parts of three different types, namely an *offset*, an *add* and two *noise* definitions.

```
Heightmap Underground = Offset(0,-10,0) Add (
        Noise(128,25)
        Noise(16,7)
)
```

Height maps can be parsed via the following parse rule.

```
Heightmap:
    | NOISE LPARAN Num COMMA Num RPARAN
      { Noise2D($3,$3,$5) }
    | NOISE LPARAN Num COMMA Num COMMA
      Num RPARAN
      { Noise2D($3,$5,$7) }
    | PLANE Num
      { Plane $2 }
    | ADD LPARAN HeightmapList RPARAN
      { Add2D $3 }
    | OFFSET Position Heightmap
      { Offset2D($2,$3) }
    | HEIGHTMAP LPARAN ID RPARAN
      { HMRef $3 }
```

The parse rule for the add type of a height map takes a list of height maps. Semantically it doesn't make sense to calculate the sum of zero height maps, so the height map list is defined as a non-empty list. A parse rule for this can be seen below.

```
HeightmapList:
    | Heightmap
      { [$1] }
    | Heightmap HeightmapList
      { $1::$2 }
```

6.6 Volume Maps

Below can be seen an example of a volume map in the game definition language.

```
Volumemap Cliffs = Add (
        Noise(32,3)
        Noise(8,3)
        Noise(2,1)
)
```

Volume maps can be parsed via the following parse rule. As expected, the parse rule for a volume map closely matches that of a height map.

```
Volumemap:
    | NOISE LPARAN Num COMMA Num RPARAN
      { Noise3D($3,$3,$3,$5) }
    | NOISE LPARAN Num COMMA Num COMMA
      Num COMMA Num RPARAN
      { Noise3D($3,$5,$7,$9) }
    | ADD LPARAN VolumemapList RPARAN
      { Add3D $3 }
    | OFFSET Position Volumemap
      { Offset3D($2,$3) }
    | VOLUMEMAP LPARAN ID RPARAN
      { VMRef $3 }
```

As with height maps, the rule for addition of multiple volume maps requires a non empty list of volume maps. This can be parsed via the following parse rule.

```
VolumemapList:
        | Volumemap
          { [$1] }
        | Volumemap VolumemapList
          { $1::$2 }
```

6.7 Landscapes

An example of a landscape in the game definition language can be seen below. Note that this landscape has a reference to the height map given as an example in section 6.5 and volume map given as an example in section 6.6.

```
Landscape Sheeponia = Heightmap (
    Heightmap(Underground)
    Volumemap(
        Volumemap(Cliffs)
        (-100,1,Voxel(Dirt))
        AirVoxel )
    Landscape(Grass) )
```

Landscapes can be parsed via the following three parse rule. Note that the *game object* rule from the conceptual model given in section 5.2.3 are explicitly defined here as being able to yield items and NPCs only. In addition, a keyword for a non-existing voxel is added, namely *airvoxel*.

```
Landscape:
    | HEIGHTMAP LPARAN Heightmap
      Landscape Landscape RPARAN
      { Heightmap($3,$4,$5) }
    | AREAMAP LPARAN Heightmap
      RangeList Landscape RPARAN
      { AreaMap($3,$4,$5) }
    | VOLUMEMAP LPARAN Volumemap
      RangeList Landscape RPARAN
      { VolumeMap($3,$4,$5) }
    | LANDSCAPE LPARAN ID RPARAN
      { LandscapeRef $3 }
    | NPC LPARAN ID RPARAN
      { Gameobject("Npc",$3) }
    | ITEM LPARAN ID RPARAN
      { Gameobject("Item",$3) }
    | STRING
      { VoxelVal (VoxelFromString $1) }
    | AIRVOXEL
      { VoxelVal airVoxel }
    | VOXEL LPARAN ID RPARAN
      { VoxelRef $3 }
```

The *area map* and *volume map* rules above make use of a range list. This is assumed to be a non-empty list. Two parse rules defining non-empty range lists and ranges can be seen below.

```
RangeList:
    | Range { [$1] }
    | Range RangeList
      { $1::$2 }
```

```
Range:
    LPARAN Num COMMA Num COMMA
    Landscape RPARAN
    { ($2,$4,$6) }
```

6.8 Levels

An example of a level in the game definition language can be seen below. It has a reference to the landscape defined in section 6.7.

```
Level CarlOfSheeponia = (
    Landscape = Sheeponia
    PlayerSpawnPoint = (0,5,0)
    Attr = () )
```

Levels can be parsed via the following parse rule.

```
LevelDef:
    LANDSCAPE EQ ID
    PLAYERSPAWNPOINT EQ Position
    ATTR EQ LPARAN NamedList RPARAN
    { {
        Landscape = $3;
        PlayerSpawnPoint = $6;
        Attr = Map.ofList $10;
    } }
```

6.9 Behavior Trees

A partial definition of a behavior tree for a player, given in the game definition language, can be seen below. Note that, relating to the semantics for the return state of a behavior tree of a player as given in section 2.2.6, the first two *decorator* nodes define the losing and winning condition for the player.

```
BehaviorTree PlayerBehavior =
  Parallel (1,1) (
    Decorator (WaitForFailure())
      Condition (Player.Attr("Health") > 0)
```

```
    Decorator (WaitForSuccess())
      Condition (Player.Attr("HasWon"))
    Decorator (InfiniteLoop())
      Parallel (-1,-1) (
        Sequence (
          Condition (Player.Input.ButtonDown("Jump"))
          Condition (Player.Character.IsGrounded())
          Action Player.MoveY (Player.Attr("Jump Speed"))
        )
        ...
      )
      ...
  )
```

Behavior trees can be parsed via the following parse rule, with the expression (Expr) and action-expression (ActionExpr) rules described in section 6.10.

```
BehaviorTree:
    | SEQUENCE LPARAN BehaviorTreeList RPARAN
      { Sequence $3 }
    | SELECTOR LPARAN BehaviorTreeList RPARAN
      { Selector $3 }
    | PARALLEL LPARAN INT COMMA INT RPARAN
      LPARAN BehaviorTreeList RPARAN
      { Parallel($3,$5,$8) }
    | DECORATOR LPARAN ActionExpr RPARAN
      BehaviorTree
      { Decorator($3,$5) }
    | LINK ID
      { Link $2 }
    | CONDITION LPARAN Expr RPARAN
      { Condition $3 }
    | ACTION ActionExpr
      { Action $2 }
```

A, possibly empty, list of behavior trees can be parse via the following parse rule.

```
BehaviorTreeList:
    | { [] }
    | BehaviorTree BehaviorTreeList
      { $1::$2 }
```

6.10 Expressions

As mentioned in section 5.4, expressions are left out of this project, but a preliminary parser, based on the preliminary conceptual model for expressions seen in appendix C.3.9, can be seen in the parser definition in appendix C.2.2.

6.11 Game Definition

Building on the definitions earlier in this chapter, the definition of a complete game can be parsed via the following parse rule. Note that all objects but the player has an associated name (ID), and the addition of a voxel rule. The *DefaultGame* object in the last line of the definition below is an empty game definition record, used as a starting point.

```
GameDefinition:
    | LEVEL ID EQ LPARAN LevelDef RPARAN GameDefinition
      { let v = $7;
        {v with Levels = v.Levels.Add($2,$5)} }
    | PLAYER EQ LPARAN PlayerDef RPARAN GameDefinition
      { let v = $6;
        {v with Player = $4} }
    | ITEM ID EQ LPARAN ItemDef RPARAN GameDefinition
      { let v = $7;
        {v with Items = v.Items.Add($2,$5)} }
    | NPC ID EQ LPARAN NpcDef RPARAN GameDefinition
      { let v = $7;
        {v with Npcs = v.Npcs.Add($2,$5)} }
    | BEHAVTREE ID EQ BehaviorTree GameDefinition
      { let v = $5;
        let bts = v.BehaviorTrees.Add($2,$4)
        {v with BehaviorTrees = bts} }
    | VOXEL ID EQ STRING GameDefinition
      { let v = $5;
        let voxel = VoxelFromString $4
        {v with Voxels = v.Voxels.Add($2, voxel)} }
    | HEIGHTMAP ID EQ Heightmap GameDefinition
      { let v = $5;
        {v with Heightmaps = v.Heightmaps.Add($2,$4)} }
    | VOLUMEMAP ID EQ Volumemap GameDefinition
      { let v = $5;
```

```
        {v with Volumemaps = v.Volumemaps.Add($2,$4)} }
    | LANDSCAPE ID EQ Landscape GameDefinition
      { let v = $5;
        {v with Landscapes = v.Landscapes.Add($2,$4)} }
    | EOF
      { DefaultGame }
```

Chapter 7

Constructing the Game Model

Now that most of the process of developing a game has been discussed, all that needs to be discussed is how to actually create a game from the different object definitions. What a game formally is and consists of, is presented in this chapter.

A few general purpose functions has been created, which can be seen in appendix C.3.3. A module that binds together the concepts from this chapter and creates a game from it can be seen in appendix C.3.12, and a class that allows Unity to make use of it can be seen in appendix C.3.13.

7.1 Visual Voxel Object

As described in chapter 4, the appearance of the landscape and game objects are represented internally as one or more chunks, and transformed to a mesh and added to a Unity game object (*GameObject*). To maintain a connection between the internal representation and Unity's rendering, a type called VisualVoxelObject is used, which can be seen below.

```
type VisualVoxelObject =
    UnityEngine.GameObject * Chunk * float32
```

The first part of the tuple is a game object as represented by Unity. The second part is the voxel data in the form of a chunk, and the third part is the scale of the voxel model, where the landscape always has a scale of 1.

Given an appearance definition of type *Map<string, PrimitiveValue>*, a VisualVoxelObject can be created with the following algorithm, which looks for certain keywords in the given appearance definition:

```
let CreateVisualObject (appearence : Map<_,_>) goTitle =
    let scale =
        match appearence.TryFind "Scale" with
        | Some (PrimFloat f) when f > 0.0f -> f
        | Some (PrimInt i) when i > 0 -> float32 i
        | _ -> 1.0f
    let chunkstr =
        match appearence.TryFind "Chunk" with
        | Some (PrimStr s) -> s
        | _ -> ""
    let voxelcolor =
        match appearence.TryFind "Color" with
        | Some (PrimStr s) -> s
        | _ -> "#0ff"
    let chartovoxel c =
        match c with
        | '1' -> Some (VoxelFromString voxelcolor)
        | '0' -> Some airVoxel
        | _ -> None
    let getdim s =
        match appearence.TryFind s with
        | Some (PrimInt i) when i > 1-> i
        | _ -> 1
    let (w,h,d) =
        (getdim "Width", getdim "Height", getdim "Depth")
    let chunkdata =
        ChunkdataFromString w h d chartovoxel chunkstr
    let centerpos : Position =
        match appearence.TryFind "Center" with
        | Some (PrimPosition (x,y,z)) -> (x,y,z)
            | _ ->
                (float32 w / 2.0f,
                    float32 h / 2.0f,
                    float32 d / 2.0f)
        (new UnityEngine.GameObject(goTitle),
            (chunkdata, centerpos),
```

```
                scale)
```

Notice that this algorithm doesn't visualize the voxel model. However, given the *ChunkToMesh* algorithm described in section 4.4, visualizing a voxel model is merely a matter of creating the meshes and assigning them to the Unity game object model.

The argument *goTitle* in the above algorithm is a string, which will be the name of the game object in Unity.

7.2 Creating Game Objects

As both items and NPCs can be considered to be simplified versions of a player character, I will here demonstrate how to create a player character only, with creation of items and NPCs following similar but simpler procedure.

To instantiate a player character object, a player definition as defined in section 5.1.3 is needed, and can be transformed to the following model:

```
type Player =
    {
    appearence : VisualVoxelObject;
    attr : Map<string, PrimitiveValue>;
    behaviorTree : BehaviorTree;
    behaviorTreeStatus : BehaviorTreeStatus;
    inventory : Map<string, Item>;
    }
```

How to create the appearance is discussed in section 7.1. Due to the immutability of the *Map* type, the attributes can be assigned directly. The same with the behavior tree. The inventory can be readily created, given the assumption that items can be readily created. The *BehaviorTreeStatus* type is related to the evaluation of a behavior tree, and is discussed more in section **??**.

The full source code for the player character, NPC and item representations can be seen in appendix C.3.11, along with algorithms to instantiate them from the definitions given in section 5.1.

7.3 Evaluate a Landscape Definition

Evaluating a height map, a volume map or a landscape is relatively straight forward. I will give an example of each of these below, and the full source code for all three can be seen in appendix C.3.7. In all three cases, it is possible to have a reference to another height map, volume map, landscape, a voxel or a game object of some type. Therefore they make use of a record with functions to retrieve the referenced values from some environment, as defined below:

```
type 'a LandscapeEnv =
    {
    GetHeightmap : string -> Heightmap option;
    GetVolumemap : string -> Volumemap option
    GetLandscape : string -> Landscape option;
    GetVoxel : string -> Voxel option;
    GetObject : string -> string -> 'a;
    }
```

Evaluation of the height maps and volume maps require some noise function, which here is a Simplex-Perlin implementation from an external source listed in appendix B.4, as seen below.

```
let internal NoiseGen =
    new Graphics.Tools.Noise.Primitive.SimplexPerlin()
```

Evaluating a height map for some (x, z) coordinate yields a y-value for those coordinates. A small sample of the implementation can be seen below.

```
let rec EvaluateHeightmap landscapeenv x z hm =
    match hm with
    | Noise2D(sx,sz,weight) when
        sx <> 0.0f && sz <> 0.0f && weight <> 0.0f ->
        NoiseGen.GetValue(x / sx, z / sz) * weight
    ...
```

Evaluating a volume map for some (x, y, z) coordinate yields a numerical value for that coordinate. A small sample of the implementation can be seen below.

```
let rec EvaluateVolumemap landscapeenv pos vm =
    match vm with
    ...
    | Add3D vms ->
        List.map
            (EvaluateVolumemap landscapeenv pos) vms
```

```
            |> List.fold (fun a b -> a + b) 0.0f
    ...
```

Evaluating a landscape yields a value of the type *LandscapeResult*, which can be seen below. This is used because the landscape procedure should not only be able to create a voxel landscape, but also some form of game objects within the landscape.

```
type 'a LandscapeResult =
    | VoxelValue of Voxel
    | Object of 'a
```

A sample of the algorithm for evaluating a landscape can be seen below.

```
let rec EvaluateLandscape landscapeenv landscape
        ((x,y,z) as pos) =
    let evalland = EvaluateLandscape landscapeenv
    match landscape with
    | Heightmap (hm, landbelow, landabove) ->
        let hmy = EvaluateHeightmap landscapeenv x z hm
        let landscape' =
            if hmy > y
            then landbelow
            else landabove
        evalland landscape' pos
    ...
```

7.4 Formal Game Model

Here follows an attempt at making a formal definition of what a game is, using the definition by Salen and Zimmerman mentioned in section 2.2.4.

The outcome can for each player only ever be *won*, *lost* or *tied*. Each of these outcomes may be applied to each individual player, which then signifies the end of the game for that player. This matches the *quantifiable outcome* from Salen & Zimmerman's definition, and conveniently also matches the three possible end-states of for instance a soccer match. Thus the outcome can be defined as follows:

```
type Result =
    | Won
```

```
    | Lost
    | Tie

type Outcome<'s, 'p> =
    's -> 'p -> Result option
```

The semantics is that, in some state, the game is over for some player if the outcome-function returns a value different from `None`, and the quantifiable outcome can then be read from the returned value. In the case of *Carl of Sheeponia*, the game is lost when the player's health reaches zero, but can never be won nor tied.

The rules can in general be put into one of three categories: Rules that apply to a single player in a certain state, rules that apply to all players, and rules that apply to just the state and no players. This can be formally defined as follows:

```
type Rule<'s, 'p> =
    | PerPlayerRule of ('s -> 'p -> 's)
    | AllPlayerRule of ('s -> ('p list) -> 's)
    | StateRule of ('s -> 's)
```

Note that in this case *AllPlayerRule* encompasses the other two rules, but for convenience for the game developer, all three rules are defined. The rules are evaluated as follows:

```
let EvaluateRule players state rule =
    match rule with
    | PerPlayerRule f ->
        List.fold f state players
    | AllPlayerRule f ->
        f state players
    | StateRule f ->
        f state
```

Note that the *PerPlayerRule* may, in cases of multiple players, give some players an advantage to exploit in that *PerPlayerRule* is always evaluated in a serial fashion in the same order. If a rule is expected to be performed in parallel for all players at the same time, *AllPlayerRule* should be used, which can either calculate the rule in parallel and handle conflicts, or calculate the rule in serial and hide this with for instance a randomization of the order.

Realizing that the state of a game changes throughout the entire execution of the game, and using the above definitions, a game can be defined as below.

```
type Game<'s, 'p> =
    'p list * (Rule<'s, 'p> list) * Outcome<'s, 'p>
```

Using this definition, the process of going from one state to the next can be calculated as follows:

```
let GameStep ((players,rules,outcome) : Game<_,_>) state =
    let GameNotEnded player =
        outcome state player = None
    let activeplayers =
        List.filter GameNotEnded players
    let EvalRule state rule =
        EvaluateRule activeplayers state rule
    match activeplayers with
    | [] -> (state, false)
    | _ -> (List.fold EvalRule state rules, true)
```

Note that there is a specific order of the rules, which can be exploited to give a priority to some rules.

The source code for this definition of a game can be seen in appendix C.3.1.

7.5 State Definition

Now that a formal definition of a game has been given, and most, if not all, individual parts of the game and the development of it has been discussed, the game state can be defined.

Recall that the evaluation of a landscape may return either a voxel or some form of an object. Using the concepts for game objects used in this project, the two types of objects that can be created in the landscape are items and NPCs, which is captured by the *ItemNpc* type below.

```
type ItemNpc =
    | ItemRes of ItemDef
    | NpcRes of NpcDef
```

The landscape of a game should be able to be partially exchanged when the player moves around, and thus the landscape should consist of multiple chunks. These are stored in a hyper-rectangular 3-dimensional array of *VisualVoxelObjects*. Besides that, the minimum (x, y, z) coordinate of the list of chunks, or the *offset* of the landscape, is stored, and a list of chunks that need to be recreated, either because the view has changed or because the chunk data has changed, as well as the dimensions of each chunk and the amount of chunks in the landscape. Furthermore a function to create a voxel (or game object) at a given position is stored. All together, these form the landscape in a game state, as seen below.

```
type Landscape =
    {
    chunks : (VisualVoxelObject option)[,,];
    landscapeOffset : Position;
    refreshChunks : Position list;
    voxelCreator : Position -> ItemNpc LandscapeResult;
    chunkDimensions : int * int * int;
    chunkAmount : int * int * int;
    }
```

Apart form the Landscape type above, the state of a game is merely a list of items, a list of NPCs, a single player character and some attributes, as defined below.

```
type State =
    {
    landscape : Landscape;
    items : Item list;
    npcs : Npc list;
    player : Player;
    attr : Map<string, PrimitiveValue>;
    }
```

7.6 Player Definition

As can be seen in the evaluation of a game in section 7.4, while the state may change over time, the player(s) remain constant. As such, the player of a formal game is here defined as a mapping between the human-computer interface and the game, as can be seen below.

```
type InputButton =
    | Key of UnityEngine.KeyCode
    | MouseButton of int
type Input =
    | Button of string * InputButton
    | Axis of string * string
```

```
type PlayerController = Input list
```

From this can be seen that input may have one of two forms: Either a button or an axis. Information about a button, for instance whether it is currently pressed, may at any time return either true or false, while an axis will yield a number between -1 and 1 depending on the state. For instance, can a 1-axis joystick be at any position between its two extreme positions.

Getting information about a button can happen via the following algorithm.

```
let PlayerButton keyf mousef controller name =
    let buttonPressed (input : Input) : bool =
        match input with
        | Button(name', inputbutton) when name' = name ->
            match inputbutton with
            | Key(kc) ->
                keyf kc
            | MouseButton(n) ->
                mousef n
        | _ ->
            false
    List.exists buttonPressed controller
```

The above algorithm requires two functions as parameters. Below are three examples of this with Unity specific functions.

```
let PlayerButtonPressed =
    PlayerButton (UnityEngine.Input.GetKey)
        (UnityEngine.Input.GetMouseButton)
let PlayerButtonDown =
    PlayerButton (UnityEngine.Input.GetKeyDown)
        (UnityEngine.Input.GetMouseButtonDown)
let PlayerButtonUp =
    PlayerButton (UnityEngine.Input.GetKeyUp)
        (UnityEngine.Input.GetMouseButtonUp)
```

The state of an axis can be fetched via the following algorithm, taking a *PlayerController* and an axis-name as parameters.

```
let PlayerAxis controller name =
    let findaxis input =
        match input with
        | Axis (name', _) ->
            name' = name
```

```
        | _ -> false
    match List.tryFind findaxis controller with
    | Some(input) ->
        match input with
        | Axis (_,name') ->
            UnityEngine.Input.GetAxis(name')
        | _ ->
            0.0f
    | _ ->
        0.0f
```

The source code for the player controller can be seen in appendix C.3.2.

7.7 Winning and Losing Conditions

As mentioned in section 2.2.6, the winning and losing conditions are here based on the result of the behavior tree for the player. Thus the outcome for a game can be defined as below.

```
let OutcomeFunction : Outcome<State, PlayerController> =
    fun (state : State) _ ->
        match state.player.behaviorTreeStatus with
        | BTSuccess -> Some Won
        | BTFail -> Some Lost
        | _ -> None
```

CHAPTER 8

Tests

At least two specific types of tests need to be accomplished on a project such as this: System tests, evaluating the system as a whole, and functional tests, evaluating the individual parts. This chapter will discuss a few examples of how these tests could be constructed.

8.1 System tests

System tests of a game development platform takes the form of taking a game definition as input, and should as output give a game following the given definition. In this project, this falls within three areas: Landscapes, game objects and behavior trees.

8.1.1 Landscape creation

The first thing to be tested here is whether a landscape definition is evaluated as expected. In figure 8.1 can be seen a landscape generated from the Carl of Sheeponia definition as given in appendix C.1.1. Based on multiple landscapes

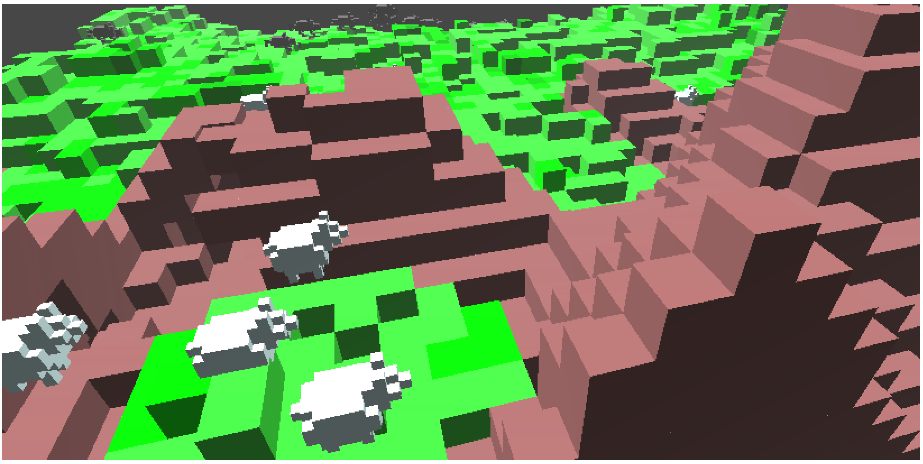


Figure 8.1: A section of the landscape of Carl of Sheeponia.

inspected through Unity's ability to inspect a game, the landscape procedure seems to work as intended.

8.1.2 Game object creation

In the landscape generated above can also be seen some sheep and wolfs. Taking a closer look at a wolf, as seen in figure 8.2, reveals that this seems to be rendered as it should.

8.1.3 Mesh creation

By making use of some of Unity's features, the game can be paused and inspected in more detail. In figure 8.3 can be seen the meshes of the wolf as seen above. This reveals that the number of polygons is minimized, as expected.

8.2 Other Tests Needed

Other than system tests, there should be some automated tests that verify the individual parts of the platform, for instance by creating the polygon mesh for a specific chunk and count the amount of created polygons. Unfortunately, none

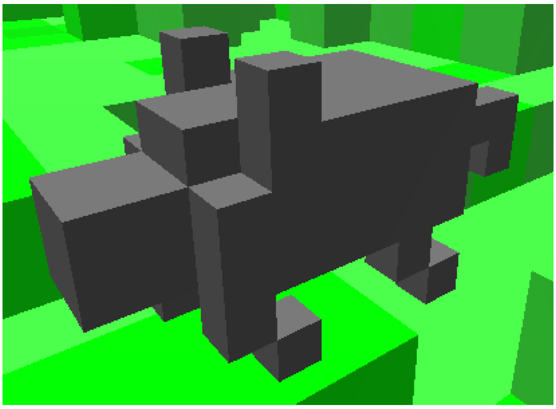


Figure 8.2: A wolf from Carl of Sheeponia.

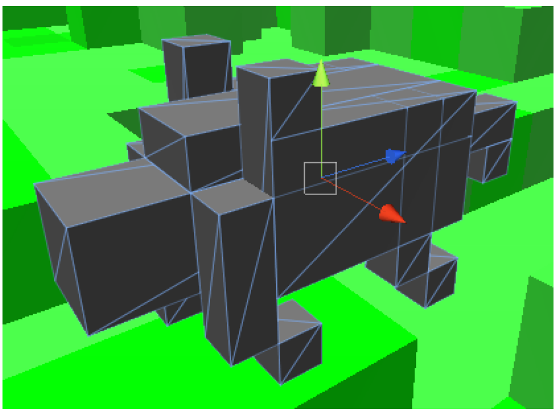


Figure 8.3: Example of the polygons a wolf consists of.

such tests has been created in this project. However, two important parts to be tested via automated tests is the scanner/parser of the game definition language, and the evaluation of behavior trees/expressions.

Chapter 9

Results

In this project, a *game definition language* has been created, which divides a game into common parts, specifically game objects consisting of items, NPCs and player characters, landscapes built using height and volume maps, and behavior of NPCs and player characters defined via behavior trees.

For transforming an instance of the game definition language into a game, a scanner/parser has been created which transforms it into a conceptual model of a game. From this conceptual model, a game can be created and formed into a formal game following the definition given in section 7.4. The rules of the created game can be modified by modifying the behavior of the NPCs and player characters in the game. A state and a player controller for the formal game model has been created. A partial transformation from a conceptual game model to a formal game model has been created.

The implementation of scripting behavior via behavior trees has shown that there is an unfulfilled need to be able to also define the actions, decorators and possibly more functions.

9.1 Landscapes

Along with a definition of a landscape, an evaluation of a landscape has been created, as well as a voxel library for displaying the landscape. An example of an incremental creation of a landscape is discussed in this section.

A natural starting point for creating a landscape is that of a plane, as can be seen below. The result of this can be seen in figure 9.1a.

```
Landscape Result1 = Heightmap (
    Plane 0.0
    "#930"
    AirVoxel
)
```

Adding some large hills to the landscape can be done by replacing the plane with a noise-function as below. The result of this can be seen in figure 9.1b.

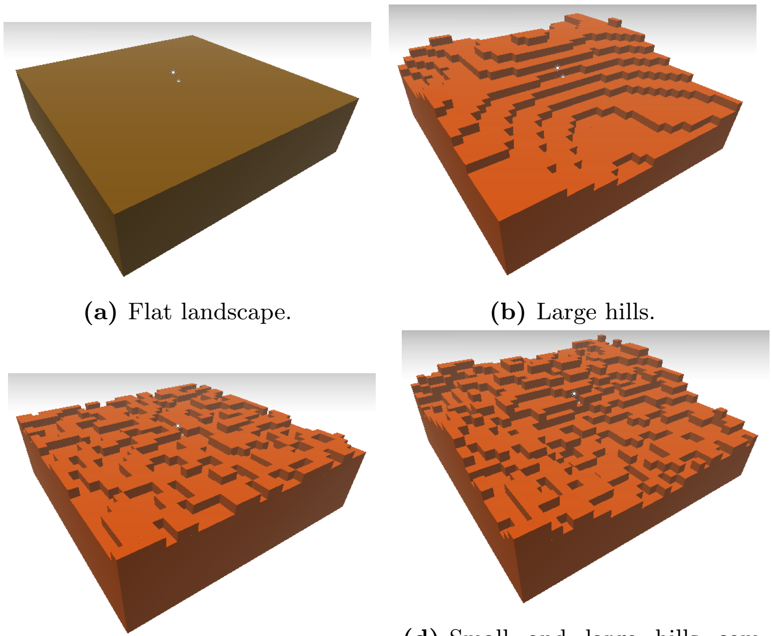
```
Landscape Result2 = Heightmap (
    Noise(32,5)
    "#930"
    AirVoxel
)
```

Instead of large hills, small hills can be created by tweaking the parameters for the noise-function, as below. The result of this can be seen in figure 9.1c.

```
Landscape Result3 = Heightmap (
    Noise(4,1)
    "#930"
    AirVoxel
)
```

Creating a more natural landscape, with a combination of the two previous noise-functions, can be done by adding them together as can be seen below. The result of this can be seen in figure 9.1d.

```
Landscape Result4 = Heightmap (
    Add(
        Noise(32,5)
        Noise(4,1)
    )
    "#930"
    AirVoxel
```



(a) Flat landscape. (b) Large hills.

(c) Small hills. (d) Small and large hills combined.

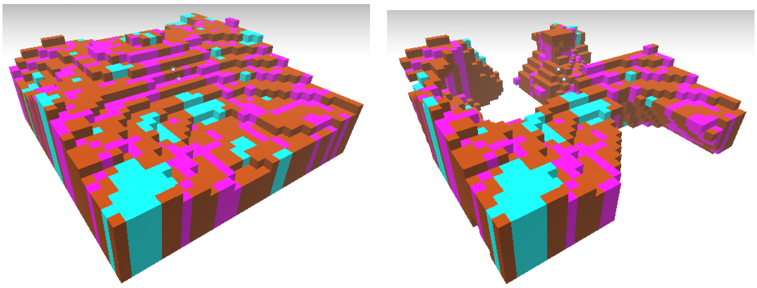
Figure 9.1: Examples of height map landscapes.

```
)
```

Building on the landscape in figure 9.1b, the landscape can be divided into areas (like countries) as below, the result of which can be seen in figure 9.2a.

```
Landscape Result5 = Heightmap (
    Noise(32,5)
    Areamap(
        Noise(8,5)
        (-1000,-3,"#0ff")
        (0,2,"#f0f")
        "#930"
    )
    AirVoxel
)
```

So far, the landscapes have been created by using height maps only. By using volume maps as well, 3-dimensional structures such as caverns can be created.



(a) Area map. (b) Area and volume map.

Figure 9.2: Examples of landscapes with area and volume maps.

An example of this can be seen below, and the result of this is displayed in figure 9.2b.

```
Landscape Result6 = Heightmap (
    Noise(32,5)
    Volumemap(
        Noise(16,5)
        (-1000,0,AirVoxel)
        Areamap(
            Noise(8,5)
            (-1000,-3,"#0ff")
            (0,2,"#f0f")
            "#930"
        )
    )
    AirVoxel
)
```

All of the examples from this section can also be found in appendix C.1.2.

Chapter 10

Discussion

The created game definition language is tailored to games, unlike most other programming languages, and while it doesn't do it yet, it is intended to cover all aspects of game development. However, it is not intended that all development should happen through scripting, as aspects such as 3-dimensional models can only be scripted to a very limited degree. Instead the idea is to create graphical editors, for which the output is in the format of the game definition language. This gives a few advantages, most notably it allows for fast prototyping of simple elements through scripting, and more advanced elements through specialized editors. And hopefully (this has not been tested) it will allow games developed with this language to be seamlessly used in version control software such as Git, SVN and CVS.

While the created game definition language is targeted towards first person games with a single player, only few elements in the language is tailored for this, and as such it should be easily extensible to other genres and multiple players. However, with multiple players on multiple computers, the underlaying architecture would have to be changed to support a client/server structure. This itself brings other problems, such as where to calculate the computations of game logic: On the server or on the client. Calculating them on the server only puts a lot of strain on the server, and calculating them on the clients only enables clients to cheat with the calculations and give the cheating users an advantage. This could potentially be mitigated by performing the calculations on the client, and have the server make random checks of the calculations performed by the clients.

Some definitions of what a game is or consists of, talk about more abstract terms such as *fun* and *challenges*. While these definitions can help understand what a game is, and help make a game, they are not very useful for creating a game development platform. For this, only a definition that can be formalized can be used, such as the one by Salen & Zimmerman which is used in this report.

Creating a landscape with the game definition language is very simple, once the semantics of the language are understood. Landscapes created in this manner are infinite, and unless specifically defined to not be, they will be continuos. Further more, the use of Simplex/Perlin noise has been used in many other projects to create textures, landscapes and more, that seem natural.

The implementation of behavior trees has shown that behavior trees can indeed be used in scripting, provided that it is still possible to create functions in an imperative or similar language. Some aspects are still lacking in this implementation though, primarily adding semantics to sub-trees in the behavior trees, for instance via specifically constructed comments.

The classification of game objects into items, NPCs and player characters may make the game development process simpler, as the game developer doesn't have to worry about what classes of objects should appear in his or her game, but this also reduces the flexibility of the solution.

While one of the arguments for creating a voxel based game development platform was the ability for the game developer to easily modify the landscape manually, this has not been implemented in this project. However, this ability has been demonstrated in other projects, in particular in the game Minecraft, in which many impressive structures has been created by the users. Furthermore it can be argued that there is a lack of voxel based game development platforms on the market.

The goal as described in section 1.1 hasn't been completely reached, as it is not possible to create a working game, but the essential components of creating a game has been identified, and some of them implemented. The primary task left is to implement an interpreter for the behavior trees, which in itself is no small task.

10.1 Future Work

All in all, it is a very large project to create a game development platform, and it has been far from finished here. Therefore this section give some remarks on some of the most important areas for further development, besides finishing what has already been started.

As mentioned, it should be possible for the game developer to create functions. For this end, it would probably be a good idea to use an existing language. A candidate could be LUA, which is a scripting language already in use in many games, and thus would reduce the entry barrier to this game development platform for some game developers.

In this project, only single player games can be created. It is prudent to mitigate this for obvious reasons.

Besides creating a WYSIWYG editor for editing the 3D-models used, it should also be possible to create composite voxel models, i.e. a single object consisting of multiple voxel models, each which can be freely rotated and positioned according to the other. The various models of a single object could be joined by hinges, which would enable animations of the objects.

The current state of creating landscapes is simple yet powerful, but adding a few tools may make it even more powerful. Specifically should be added multiplication of multiple height maps or volume maps, using a height map to create a gradient in volume maps, and adding voxel models to the landscape (as a part of the landscape, not as game objects) after the landscape has been created, which would enable trees and cities to be a part of the landscape.

A very important addition is to be able to serialize and deserialize the state of a game, such that a game may be stopped and resumed at any time the human player wishes. Currently, the game has to be played from start to end in one sitting.

It should be possible to add sounds to the game. Adding sounds would be required to be as external resources, as it is not expected that it is possible to create sounds from scratch via scripts.

Appendix A

Glossary

FSLex Transforms a definition of a lexer into F#. Part of the F# PowerPack.

FSYacc Transforms a definition of a parser into F#. Part of the F# PowerPack.

Polygon A single triangle located in 3-dimensional space.

Polygon count A metric used for optimizing the speed at which games can run, by determining how many polygons some model consists of.

Polygon mesh A collection of polygons that forms one or more surfaces.

Voxel Contraction of *volumetric pixel*. A point in a 3-dimensional grid with an associated value.

WYSIWYG What You See Is What You Get - a class of graphical editors, where the output is the same or very close to the same as seen in the editor.

Appendix B

External Sources

This appendix contains the sources of images from external sources, and a library used in the code.

B.1 Wireframe Character

Image from an external source. Per 26/8/2013:
http://www.rocketbox-libraries.com/index.php/characters/complete-characters/cc-sportive-03-f.html
http://www.rocketbox-libraries.com/media/catalog/product/cache/1/image/5e06319eda06f020e43594a9c230972d/s/p/sportive03_f_lods_wire0000.jpg

B.2 Voxel Character

Image from an external source. Per 26/8/2013:
https://picroma.com/cubeworld
https://picroma.com/images/Avatar.jpg

B.3 Minecraft Landscape

Image from an external source. Per 28/8/2013:
http://gamesminecraft.org/landscape/
http://gamesminecraft.org/games/img/landscape-minecraft-2560x1600-wallpaper.jpg

B.4 Simplex/Perlin Noise

The noise-library used in this project comes from an external source, namely *LibNoise for .NET*: http://libnoisedotnet.codeplex.com/. This is a .NET port of another library, namely *libnoise*: http://sourceforge.net/projects/libnoise/.

Appendix C

Source Code

This appendix contains the source code for the created game development platform, including examples.

C.1 Game Definition Language

In this section are two examples of the *game definition language* created in this project.

C.1.1 Carl of Sheeponia

```
Item Sword =
(
    Appearance =
    (
        "Width" = 1
        "Height" = 8
        "Depth" = 3
        "Color" = "#B83"
```

```
        "Chunk" = "010 010 111 010 010 010 010 010"
        "Center" = (0.5, 1.5, 1.0)
        "Scale" = 0.2
    )
    Attr =
    (
        "Weapon Type" = "Melee"
        "Attack Distance" = 2.5
        "Attack Strength" = 15
        "Attack Speed" = 1.2
    )
)

Item Rifle =
(
    Appearance =
    (
        "Width" = 1
        "Height" = 3
        "Depth" = 8
        "Color" = "#ABA"
        "Chunk" = "001 001 101 011 001 110 001 001"
        "Center" = (0.5, 5.5, 1.0)
        "Scale" = 0.2
    )
    Attr =
    (
        "Weapon Type" = "Melee"
        "Attack Distance" = 200
        "Attack Strength" = 40
        "Attack Speed" = 1.4
        "Bullets In Magazine" = 5
        "Max Bullets In Magazine" = 5
        "Reload Speed" = 2.1
    )
)

Player =
(
    Appearance =
    (
        "Width" = 5
        "Height" = 10
        "Depth" = 3
```

```
        "Color" = "#934"
        "Chunk" = "01010 00000 00000 01110 01110 01110
            00000 00000 00100 00000  01010 01010 01010
            01110 11111 11111 00100 01110 01110 01110
            00000 00000 00000 01110 01110 01110 00000
            00000 01110 01110"
        "Scale" = 0.2
    )
    Attr =
    (
        "Health" = 100
        "Bullets Left" = 50
        "Weapon Switch Time" = 1
        "HasWon" = false
        "Jump Speed" = 1
        "Turn Speed" = 1
        "Move Speed" = 1
        "Strafe Speed" = 1
        "Look Speed" = 1
    )
    BehaviorTree = PlayerBehavior
    Camera =
    (
        "Center" = (0,-1,0)
        "Facing" = "Forward"
    )
    Inventory =
    (
        "Weapon" = Sword
        "RangedWeapon" = Rifle
    )
)

NPC Sheep =
(
    Appearance =
    (
        "Width" = 5
        "Height" = 8
        "Depth" = 10
        "Color" = "#DFF"
        "Chunk" = "00000 00000 00000 00000 00000 00100
            00000 00000  00000 00000 00000 00000 00100
            01110 00100 00000  00000 00000 00000 00000
```

```
            01110 01110 01110 01010  00000 00000 01110
            11111 11111 01110 01110 00000  01010 01010
            11111 11111 11111 11111 00000 00000  00000
            00000 11111 11111 11111 11111 00000 00000
            00000 00000 11111 11111 11111 11111 00000
            00000  01010 01010 11111 11111 11111 11111
            00000 00000  00000 00000 01110 11111 11111
            01110 00000 00000  00000 00000 00000 00000
            00100 00000 00000 00000"
        "Scale" = 0.2
    )
    Attr =
    (
        "Health" = 40
        "Far Safe Distance" = 40
        "Close Safe Distance" = 10
        "Turn Speed" = 1
        "Eat Time" = 2
        "Move Speed" = 1
        "Run Speed" = 2
        "Food" = "#8D6"
        "Jump Speed" = 2
        "Die Time" = 25
        "Safe Amount" = 5
        "Run Time" = 3
    )
    BehaviorTree = SheepBehavior
)

NPC Wolf =
(
    Appearance =
    (
        "Width" = 4
        "Height" = 6
        "Depth" = 11
        "Color" = "#555"
        "Chunk" = "0000 0000 0110 0110 0000 0000  0000
            0000 0110 0110 0000 0000  1000 0001 1111 1111
            0110 0000  0001 1000 1111 1111 1111 1001  0000
             0000 1111 1111 1111 0000  0000 0000 1111 1111
             1111 0000  1000 0001 1111 1111 1111 0000
            0001 1000 1111 1111 1111 0000  0000 0000 0110
            0110 0000 0000  0000 0000 0000 0010 0000 0000
```

```
                0000 0000 0110 0010 0000 0000"
        "Scale" = 0.2
    )
    Attr =
    (
        "Health" = 70
        "Run Speed" = 2.5
        "Move Speed" = 0.8
        "Turn Speed" = 0.9
        "Jump Speed" = 1.5
        "Wolf Hunt Distance" = 30
        "Attack Distance" = 2
        "Attack Strength" = 25
        "Run Time" = 2
        "Sheep Hunt Distance" = 50
        "Eat Time" = 15
        "Health Bonus For Eating" = 50
        "Attack Health Depletion" = 5
        "Pack Size Needed To Hunt" = 3
    )
    BehaviorTree = WolfBehavior
)


Heightmap GrassColor = Add (
    Noise(32,2)
    Noise(8,1)
    Noise(2,1)
)

Landscape GrassColor = AreaMap (
    Heightmap(GrassColor)
    (-5, -2, "#090")
    (-1, 1, "#3d3")
    (2, 5, "#0a0")
    "#0f0"
)

Voxel Dirt = "#855"

Heightmap Underground = Offset(0,-10,0) Add (
    Noise(128,25)
    Noise(16,7)
)
```

```

Heightmap Grass = Offset(-13,0,19) Add (
    Noise(64,7)
    Noise(32,2)
    Noise(4,0.8)
)

Heightmap NPCs = Add (
    Noise(128,5)
    Noise(64,1)
)

Volumemap NPCs = Add (
    Noise(32,1)
    Noise(16,1)
    Noise(4,1)
    Noise(1,2)
)

Landscape Wolf = Heightmap (
    Offset(0,1,0) Heightmap(Grass)
    Volumemap (
        Volumemap(NPCs)
        (3,5,NPC(Wolf))
        AirVoxel
    )
    AirVoxel
)

Landscape Sheep = Heightmap (
    Offset(0,1,0) Heightmap(Grass)
    Volumemap (
        Volumemap(NPCs)
        (2,5,NPC(Sheep))
        AirVoxel
    )
    AirVoxel
)

Volumemap Cliffs = Add (
    Noise(32,3)
    Noise(8,3)
    Noise(2,1)
)
```

```

Landscape Grass = Heightmap (
    Heightmap(Grass)
    Landscape(GrassColor)
    AreaMap (
        Heightmap(NPCs)
        (−2.5, −1, Landscape(Wolf))
        (2, 1000, Landscape(Sheep))
        AirVoxel
    )
)

Landscape Sheeponia = Heightmap (
    Heightmap(Underground)
    Volumemap(
        Volumemap(Cliffs)
        (−100,1,Voxel(Dirt))
        AirVoxel
    )
    Landscape(Grass)
)

Level CarlOfSheeponia = (
    Landscape = Sheeponia
    PlayerSpawnPoint = (0,5,0)
    Attr = ()
)


BehaviorTree PlayerReload =
Decorator (LockResource("Weapon")) Sequence (
        Condition (Player.Inventory("Weapon").Attr("
            Weapon Type") = "Ranged")
        Condition (Player.Inventory("Weapon").Attr("
            Bullets In Magazine") < Player.Inventory("
            Weapon").Attr("Max Bullets In Magazine"))
        Condition (Player.Attr("Bullets Left") > 0)
        Action Wait (Player.Inventory("Weapon").Attr("
            Reload Speed"))
        Action Player.Attr.Add("Bullets Left", Player.
            Inventory("Weapon").Attr("Bullets In Magazine
            "))
        Action Player.Inventory("Weapon").Attr.Set ("
            Bullets In Magazine", min(Player.Inventory("
```

```
Weapon").Attr("Max Bullets In Magazine"), Player.Attr("Bullets Left")))
        Action Player.Attr.Add ("Bullets Left", - Player.Inventory("Weapon").Attr("Bullets In Magazine"))
    )

BehaviorTree PlayerAttack =
    Decorator (LockResource("Weapon")) Sequence (
        Selector (
            Condition (Player.Inventory("Weapon").Attr("Weapon Type") = "Melee")
            Condition (Player.Inventory("Weapon").Attr("Bullets In Magazine") > 0)
        )
        Action Player.Attack (Player.Inventory("Weapon").Attr("Attack Distance"), Player.Inventory("Weapon").Attr("Attack Strength"))
        Action Wait (Player.Inventory("Weapon").Attr("Attack Speed"))
        Selector (
            Condition (Player.Inventory("Weapon").Attr("Weapon Type") = "Melee")
            Action Player.Inventory("Weapon").Attr.Add ("Bullets In Magazine", -1)
        )
    )

BehaviorTree PlayerRangedWeapon =
    Decorator (LockResource("Weapon")) Sequence (
        Action Wait (Player.Attr("Weapon Switch Time"))
        Action Unrealize (Player.Inventory("Weapon"))
        Action Player.Inventory.Set ("Weapon", Player.Inventory("Ranged Weapon"))
        Action Player.Realize (Player.Inventory("Weapon"))
    )

BehaviorTree PlayerMeleeWeapon =
    Decorator (LockResource("Weapon")) Sequence (
        Action Wait (Player.Attr("Weapon Switch Time"))
        Action Unrealize (Player.Inventory("Weapon"))
        Action Player.Inventory.Set ("Weapon", Player.Inventory("Melee Weapon"))
```

```
        Action Player.Realize(Player.Inventory("Weapon"))
    )

BehaviorTree PlayerBehavior =
    Parallel (1,1) (
        Decorator (WaitForFailure()) Condition (Player.Attr("Health") > 0)
        Decorator (WaitForSuccess()) Condition (Player.Attr("HasWon"))
        Decorator (InfiniteLoop()) Parallel (-1,-1) (
            Sequence (
                Condition (Player.Input.ButtonDown("Jump"))
                Condition (Player.Character.IsGrounded())
                Action Player.MoveY (Player.Attr("Jump Speed"))
            )

            Action Player.Camera.RotateZ (Player.Attr("Look Speed") * Player.Input.Axis("Look"))
            Action Player.RotateY (Player.Attr("Turn Speed") * Player.Input.Axis("Turn"))
            Action Player.MoveX (Player.Attr("Move Speed") * Player.Input.Axis("Forward"))
            Action Player.MoveZ (Player.Attr("Strafe Speed") * Player.Input.Axis("Strafe"))
        )

        Decorator (InfiniteLoop()) Parallel (-1,-1) (
            Sequence (
                Condition (Player.Input.Button("Attack"))
                Link PlayerAttack
            )

            Sequence (
                Condition (Player.Input.Button("Reload"))
                Link PlayerReload
            )

            Sequence (
                Condition (Player.Input.Button("Ranged Weapon"))
                Condition (Player.Inventory("Weapon").Attr("Weapon Type") = "Melee")
```

```
                    Link PlayerRangedWeapon
                )

                Sequence (
                    Condition (Player.Input.Button("Melee Weapon"))
                    Condition (Player.Inventory("Weapon").Attr("Weapon Type") = "Ranged")
                    Link PlayerMeleeWeapon
                )
            )
        )



BehaviorTree SheepFeelSafe =
    Decorator (MinTimeBetween(2)) Sequence (
        Condition (NPC.FindNearest("Wolf").Distance() > NPC.Attr("Close Safe Distance"))
        Selector (
            Condition (NPC.FindNearest("Wolf").Distance() > NPC.Attr("Far Safe Distance"))
            Condition (3 * NPC.CountWithin("Wolf", NPC.Attr("Far Safe Distance")) < NPC.CountWithin("Sheep", NPC.Attr("Far Safe Distance")))
        )
    )

BehaviorTree SheepBehavior =
    Sequence (
        Selector (
            Parallel (-1,1) (
                Decorator (WaitForFailure()) Condition (NPC.Attr("Health") > 0)
                Decorator (InfiniteLoop()) Selector (
                    Parallel (-1,1) (
                        Decorator (WaitForFailure()) Link SheepFeelSafe
                        Decorator (InfiniteLoop()) Sequence (
                            Action NPC.RotateYForTime (Random(-1,1) * NPC.Attr("Turn Speed") / 2, Random(
```

```
                                NPC.Attr("Eat Time") / 2, NPC.Attr("Eat Time")))
                            Decorator (RunForTime(Random(NPC.Attr("Eat Time"), 2 * NPC.Attr("Eat Time")))) Action NPC.MoveX (NPC.Attr("Move Speed"))
                            Condition (NPC.Character.IsGrounded())
                            Condition (NPC.Attr("Food") = NPC.Character.GroundType())
                            Action Wait (Random(NPC.Attr("Eat Time"), 2 * NPC.Attr("Eat Time")))
                        )
                    )

                    Decorator (FailWhenDone()) Parallel (1,-1) (
                        Decorator (WaitForSuccess()) Link SheepFeelSafe
                        Decorator (InfiniteLoop()) Action NPC.MoveX (NPC.Attr("Run Speed"))
                        Decorator (InfiniteLoop()) Action NPC.RotateYForTime (Random(-1,1) * NPC.Attr("Turn Speed"), Random(NPC.Attr("Run Time"), NPC.Attr("Run Time") * 2))
                        Decorator (InfiniteLoop()) Decorator (MinTimeBetween(Random(1,2))) Sequence (
                            Condition (NPC.Character.IsGrounded())
                            Action NPC.MoveY (NPC.Attr("Jump Speed"))
                        )
                    )

                    Parallel (1,-1) (
                        Decorator (WaitForSuccess()) Condition (NPC.CountWithin("Sheep", NPC.Attr("Close Safe
```

```
                                Distance")) >= NPC.Attr("Safe Amount"))
                            Decorator (InfiniteLoop()) Sequence (
                                Action NPC.RotateYTowards (NPC.Attr("Turn Speed"), NPC.Nearest("Sheep"))
                                Decorator (RunForTime(NPC.Attr("Eat Time"))) Action NPC.MoveX ((NPC.Attr("Move Speed") + NPC.Attr("Run Speed")) / 2)
                            )
                        )
                    )
                )
            )
            Decorator (ReverseResult()) Action Wait (NPC.Attr("Die Time"))
        )

BehaviorTree WolfBehavior =
    Parallel (-1,1) (
        Decorator (WaitForFailure()) Condition (NPC.Attr("Health") > 0)
        Decorator (InfiniteLoop()) Selector (

            Decorator (WaitForFailure()) Sequence (
                Condition (NPC.Nearest("Wolf").Distance() > NPC.Attr("Wolf Hunt Distance"))
                Action NPC.RotateYTowards (NPC.Attr("Turn Speed"), NPC.Nearest("Wolf"))
                Decorator (RunForTime(Random(NPC.Attr("Run Time"), NPC.Attr("Run Time") * 2))) Action NPC.MoveX ((NPC.Attr("Move Speed") + NPC.Attr("Run Speed")) / 2)
            )

            Decorator (WaitForFailure()) Sequence (
                Condition (NPC.Nearest("Sheep") > NPC.Attr("Sheep Hunt Distance"))
                Action NPC.RotateYTowards (NPC.Attr("Turn Speed"), NPC.Nearest("Sheep"))
```

```
            Decorator (RunForTime(Random(NPC.Attr("Run Time"), NPC.Attr("Run Time") * 2))) Action NPC.MoveX (NPC.Attr("Move Speed"))
        )

        Condition (NPC.CountWithin("Wolf", NPC.Attr("Wolf Hunt Distance")) < NPC.Attr("Pack Size Needed To Hunt"))

        Decorator (ReverseResult()) Parallel (1,1) (
            Decorator (WaitForSuccess()) Sequence (
                Condition (NPC.Nearest("Sheep").Distance() < NPC.Attr("Attack Distance"))
                Condition (NPC.Nearest("Sheep").Attr("Health") <= 0)
            )
            Decorator (WaitForFailure()) Condition (NPC.Nearest("Sheep").Distance() < NPC.Attr("Wolf Hunt Distance"))
            Decorator (InfiniteLoop()) Sequence (
                Action NPC.MoveX (NPC.Attr("Run Speed"))
                Decorator (MinTimeBetween(Random(2,5))) Sequence (
                    Condition (NPC.Character.IsGrounded())
                    Action NPC.MoveY (NPC.Attr("Jump Speed"))
                )
            )
            Decorator (InfiniteLoop()) Sequence (
                Condition (NPC.Nearest("Wolf").Distance() <= NPC.Attr("Attack Distance"))
                Action NPC.Attack (NPC.Attr("Attack Distance"), NPC.Attr("Attack Strength"))
                Action NPC.Attr.Add ("Health", - NPC.Attr("Attack Health Depletion"))
            )
        )

```

```
            Action Wait (NPC.Attr("Eat Time"))
            Action NPC.Attr.Add ("Health", NPC.Attr("
                Health Bonus For Eating"))

        )
    )
```

C.1.2 Landscape Examples

```
Landscape Result1 = Heightmap (
    Plane 0.0
    "#930"
    AirVoxel
)

Level Result1 = (
    Landscape = ResultOneLand
    PlayerSpawnPoint = (0,0,0)
    Attr = ()
)

Landscape Result2 = Heightmap (
    Noise(32,5)
    "#930"
    AirVoxel
)

Level Result2 = (
    Landscape = Result2
    PlayerSpawnPoint = (0,0,0)
    Attr = ()
)

Landscape Result3 = Heightmap (
    Noise(4,1)
    "#930"
    AirVoxel
)

Level Result3 = (
    Landscape = Result3
    PlayerSpawnPoint = (0,0,0)
```

```
    Attr = ()
)

Landscape Result4 = Heightmap (
    Add(
        Noise(32,5)
        Noise(4,1)
    )
    "#930"
    AirVoxel
)

Level Result4 = (
    Landscape = Result4
    PlayerSpawnPoint = (0,0,0)
    Attr = ()
)

Landscape Result5 = Heightmap (
    Noise(32,5)
    Areamap(
        Noise(8,5)
        (-1000,-3,"#0ff")
        (0,2,"#f0f")
        "#930"
    )
    AirVoxel
)

Level Result5 = (
    Landscape = Result5
    PlayerSpawnPoint = (0,0,0)
    Attr = ()
)

Landscape Result6 = Heightmap (
    Noise(32,5)
    Volumemap(
        Noise(16,5)
        (-1000,0,AirVoxel)
        Areamap(
            Noise(8,5)
            (-1000,-3,"#0ff")
            (0,2,"#f0f")
```

```
                "#930"
            )
        )
        AirVoxel
)

Level Result6 = (
    Landscape = Result6
    PlayerSpawnPoint = (0,0,0)
    Attr = ()
)


Player =
(
    Appearance =
    (
        "Width" = 1
        "Height" = 1
        "Depth" = 1
        "Color" = "#934"
        "Chunk" = "1"
        "Scale" = 0.5
    )
    Attr = ()
    BehaviorTree = PlayerBehavior
    Camera = ()
    Inventory = ()
)

BehaviorTree PlayerBehavior = Action Move()
```

C.2 Scanner/Parser

In this section are the lexer and parser definitions, which are input to FSLex and FSYacc respectively, from the F# PowerPack.

C.2.1 Lexer definition

```
{
open System
open Game.BTGparser
open Microsoft.FSharp.Text.Lexing

let keywords =
    [
        "BehaviorTree", BEHAVTREE;
        "Player", PLAYER;
        "Item", ITEM;
        "Npc", NPC;
        "NPC", NPC;
        "Level", LEVEL;
        "Voxel", VOXEL;
        "Inventory", INVENTORY;
        "Heightmap", HEIGHTMAP;
        "HeightMap", HEIGHTMAP;
        "Volumemap", VOLUMEMAP;
        "VolumeMap", VOLUMEMAP;
        "AreaMap", AREAMAP;
        "Areamap", AREAMAP;
        "Plane", PLANE;
        "Landscape", LANDSCAPE;
        "Sequence", SEQUENCE;
        "Parallel", PARALLEL;
        "Selector", SELECTOR;
        "Decorator", DECORATOR;
        "Action", ACTION;
        "Condition", CONDITION;
        "Link", LINK;
        "Offset", OFFSET;
        "Add", ADD;
        "Noise", NOISE;
        "AirVoxel", AIRVOXEL;
        "Airvoxel", AIRVOXEL;
        "PlayerSpawnPoint", PLAYERSPAWNPOINT;
        "Playerspawnpoint", PLAYERSPAWNPOINT;
        "Attr", ATTR;
        "Appearance", APPEARANCE;
        "Camera", CAMERA;
        "true", TRUE;
        "false", FALSE;
        "Trim", TRIM;
```

```
            "Sqrt", SQRT;
            "Abs", ABSOLUTE;
            "Contains", CONTAINS;
            "Max", MAX;
            "Min", MIN;
            "&", AND;
            "and", AND;
            "|", OR;
            "or", OR;
        ] |> Map.ofList

let ops =
        [
            "=",    EQ;
            "<",    LT;
            "<=",   LE;
            ">",    GT;
            ">=",   GE;
            "+",    PLUS;
            "-",    MINUS;
            "*",    MULTIPLY;
            "/",    DIVIDE;
            "(",    LPARAN;
            ")",    RPARAN;
            ",",    COMMA;
            ";",    SEMI;
            ".",    DOT;
            "!",    LOGICNEG;
        ] |> Map.ofList
}

let char        = ['a'-'z' 'A'-'Z']
let digit       = ['0'-'9']
let int         = '-'?digit+
let float       = '-'?digit+ '.' digit+
let identifier  = char(char|digit|['-' '_'])*
let string      = '\"' (char|digit|['-' '_' '.' ',' ';' ':' '#' ' ' '!' '/' '\\'])+ '\"'
let whitespace  = [' ' '\t']
let newline     = "\n\r" | '\n' | '\r'
let operator    = ">" | ">=" | "<" | "<=" | "=" | "+" | "-" | "*" | "/" | "{" | "}" | "(" | ")" | "," | ";" | "." | "!"

```

```
rule tokenize = parse
| whitespace     { tokenize lexbuf }
| newline        { lexbuf.EndPos <- lexbuf.EndPos.NextLine; tokenize lexbuf; }
| int            { INT(Int32.Parse(LexBuffer<_>.LexemeString lexbuf)) }
| float          { FLOAT(Double.Parse(LexBuffer<_>.LexemeString lexbuf) |> float32) }
| operator       { ops.[LexBuffer<_>.LexemeString lexbuf] }
| identifier     { match keywords.TryFind(LexBuffer<_>.LexemeString lexbuf) with
                   | Some(token) -> token
                   | None -> ID(LexBuffer<_>.LexemeString lexbuf) }
| string         { let s = LexBuffer<_>.LexemeString lexbuf;
                   STRING(s.Substring(1,s.Length-2)) }
| eof            { EOF }
| _              { failwithf "Unrecognized input: '%s'" (LexBuffer<_>.LexemeString lexbuf) }
```

C.2.2 Parser Definition

```
%{
open Game.Base
open Game.Voxel.Voxel
open Game.Voxel.ProceduralGenerator
open Game.BehaviorTree
open Game.Prefab
%}

%token <string> ID
%token <string> STRING
%token <int> INT
%token <float32> FLOAT

%token BEHAVTREE PLAYER ITEM NPC LEVEL VOXEL LANDSCAPE INVENTORY
%token SEQUENCE PARALLEL SELECTOR DECORATOR ACTION CONDITION LINK
%token OFFSET ADD NOISE HEIGHTMAP VOLUMEMAP AREAMAP PLANE
```

```
%token ATTR APPEARANCE CAMERA
%token AIRVOXEL
%token PLAYERSPAWNPOINT
%token LPARAN RPARAN
%token COMMA SEMI DOT
%token TRUE FALSE
%token LOGICNEG TRIM SQRT ABSOLUTE
%token EQ LT LE GT GE
%token MULTIPLY DIVIDE CONTAINS
%token PLUS MINUS
%token MAX MIN AND OR
%token EOF

// start
%start GameDefinition
%type <GameDef> GameDefinition

%%

GameDefinition:
      | LEVEL ID EQ LPARAN LevelDef RPARAN GameDefinition
        { let v = $7;
          {v with Levels = v.Levels.Add($2,$5)} }
      | PLAYER EQ LPARAN PlayerDef RPARAN GameDefinition
        { let v = $6;
          {v with Player = $4} }
      | ITEM ID EQ LPARAN ItemDef RPARAN GameDefinition
        { let v = $7;
          {v with Items = v.Items.Add($2,$5)} }
      | NPC ID EQ LPARAN NpcDef RPARAN GameDefinition
        { let v = $7;
          {v with Npcs = v.Npcs.Add($2,$5)} }
      | BEHAVTREE ID EQ BehaviorTree GameDefinition
        { let v = $5;
          let bts = v.BehaviorTrees.Add($2,$4)
          {v with BehaviorTrees = bts} }
      | VOXEL ID EQ STRING GameDefinition
        { let v = $5;
          let voxel = VoxelFromString $4
          {v with Voxels = v.Voxels.Add($2, voxel)} }
      | HEIGHTMAP ID EQ Heightmap GameDefinition
        { let v = $5;
          {v with Heightmaps = v.Heightmaps.Add($2,$4)} }
      | VOLUMEMAP ID EQ Volumemap GameDefinition
```

```
        { let v = $5;
           {v with Volumemaps = v.Volumemaps.Add($2,$4)} }
      | LANDSCAPE ID EQ Landscape GameDefinition
        { let v = $5;
           {v with Landscapes = v.Landscapes.Add($2,$4)} }
      | EOF
        { DefaultGame }

LevelDef:
     LANDSCAPE EQ ID
     PLAYERSPAWNPOINT EQ Position
     ATTR EQ LPARAN NamedList RPARAN
     { {
           Landscape = $3;
           PlayerSpawnPoint = $6;
           Attr = Map.ofList $10;
     } }

Position:
     LPARAN Num COMMA Num COMMA Num RPARAN
     { ($2,$4,$6) }

ItemList:
     | { [] }
     | STRING EQ ID ItemList
       { ($1,$3)::$4 }

NamedList:
     | { [] }
     | STRING EQ PrimitiveValue NamedList
       { ($1,$3)::$4 }

PrimitiveValue:
     | INT      { PrimInt $1 }
     | FLOAT    { PrimFloat $1 }
     | STRING   { PrimStr $1 }
     | TRUE     { PrimBool true }
     | FALSE    { PrimBool false }
     | Position { PrimPosition $1 }

ItemDef:
     APPEARANCE EQ LPARAN NamedList RPARAN
     ATTR EQ LPARAN NamedList RPARAN
     { {
```

```
        Appearance = Map.ofList $4;
        Attr = Map.ofList $9;
    } }

NpcDef:
    APPEARANCE EQ LPARAN NamedList RPARAN
    ATTR EQ LPARAN NamedList RPARAN
    BEHAVTREE EQ ID
    { {
        Appearance = Map.ofList $4;
        Attr = Map.ofList $9;
        BehaviorTree = $13;
    } }

PlayerDef:
    APPEARANCE EQ LPARAN NamedList RPARAN
    ATTR EQ LPARAN NamedList RPARAN
    BEHAVTREE EQ ID
    CAMERA EQ LPARAN NamedList RPARAN
    INVENTORY EQ LPARAN ItemList RPARAN
    { {
        Appearance = Map.ofList $4;
        Attr = Map.ofList $9;
        BehaviorTree = $13;
        Camera = Map.ofList $17;
        Inventory = Map.ofList $22;
    } }

BehaviorTree:
    | SEQUENCE LPARAN BehaviorTreeList RPARAN
      { Sequence $3 }
    | SELECTOR LPARAN BehaviorTreeList RPARAN
      { Selector $3 }
    | PARALLEL LPARAN INT COMMA INT RPARAN
      LPARAN BehaviorTreeList RPARAN
      { Parallel($3,$5,$8) }
    | DECORATOR LPARAN ActionExpr RPARAN
      BehaviorTree
      { Decorator($3,$5) }
    | LINK ID
      { Link $2 }
    | CONDITION LPARAN Expr RPARAN
      { Condition $3 }
    | ACTION ActionExpr
```

```
      { Action $2 }

BehaviorTreeList:
    | { [] }
    | BehaviorTree BehaviorTreeList
      { $1::$2 }

Term:
    | STRING
      { String $1 }
    | INT
      { Num (float32 $1) }
    | FLOAT
      { Num $1 }
    | TRUE
      { Bool true }
    | FALSE
      { Bool false }
    | ActionExpr
      { Act $1 }
    | LPARAN Expr RPARAN
      { $2 }
    | UnOp Term
      { UnaryOp($1,$2) }
    | NaryOp LPARAN ExprList RPARAN
      { NaryOp($1,$3) }

Expr:
    | Term { $1 }
    | Term BinOp Expr
      { BinaryOp($1,$2,$3) }

ExprList:
    | Expr COMMA ExprList
      { $1::$3 }
    | Expr
      { [$1] }
    | { [] }

ActionRef:
    | ActionId
      { ObjectRef $1 }
    | ActionId LPARAN ExprList RPARAN
      { InvokeAction($1,$3) }
```

```

ActionExpr:
    | ActionRef DOT ActionExpr
      { MemberRef($1,$3) }
    | ActionId LPARAN ExprList RPARAN
      { InvokeAction($1,$3) }

ActionId:
    | ATTR       { "Attr" }
    | PLAYER     { "Player" }
    | NPC        { "Npc" }
    | ITEM       { "Item" }
    | INVENTORY  { "Inventory" }
    | CAMERA     { "Camera" }
    | APPEARANCE { "Appearance" }
    | ADD        { "Add" }
    | ID         { $1 }

UnOp:
    | MINUS     { NumNeg }
    | LOGICNEG  { LogicNeg }
    | TRIM      { Trim }
    | SQRT      { Sqrt }
    | ABSOLUTE  { Absolute }

BinOp:
    | EQ        { Eq }
    | LT        { Less }
    | LE        { LessEq }
    | GT        { Greater }
    | GE        { GreaterEq }
    | PLUS      { Plus }
    | MINUS     { Subtract }
    | MULTIPLY  { Multiply }
    | DIVIDE    { Divide }
    | CONTAINS  { StrContains }

NaryOp:
    | MAX { Max }
    | MIN { Min }
    | AND { And }
    | OR  { Or }

Num:
```

```
      | INT    { $1 |> float32 }
      | FLOAT { $1 }

Landscape:
      | HEIGHTMAP LPARAN Heightmap
        Landscape Landscape RPARAN
        { Heightmap($3,$4,$5) }
      | AREAMAP LPARAN Heightmap
        RangeList Landscape RPARAN
        { AreaMap($3,$4,$5) }
      | VOLUMEMAP LPARAN Volumemap
        RangeList Landscape RPARAN
        { VolumeMap($3,$4,$5) }
      | LANDSCAPE LPARAN ID RPARAN
        { LandscapeRef $3 }
      | NPC LPARAN ID RPARAN
        { Gameobject("Npc",$3) }
      | ITEM LPARAN ID RPARAN
        { Gameobject("Item",$3) }
      | STRING
        { VoxelVal (VoxelFromString $1) }
      | AIRVOXEL
        { VoxelVal airVoxel }
      | VOXEL LPARAN ID RPARAN
        { VoxelRef $3 }

RangeList:
      | Range { [$1] }
      | Range RangeList
        { $1::$2 }

Range:
     LPARAN Num COMMA Num COMMA
     Landscape RPARAN
     { ($2,$4,$6) }

Heightmap:
      | NOISE LPARAN Num COMMA Num RPARAN
        { Noise2D($3,$3,$5) }
      | NOISE LPARAN Num COMMA Num COMMA
        Num RPARAN
        { Noise2D($3,$5,$7) }
      | PLANE Num
        { Plane $2 }
```

```
        | ADD LPARAN HeightmapList RPARAN
          { Add2D $3 }
        | OFFSET Position Heightmap
          { Offset2D($2,$3) }
        | HEIGHTMAP LPARAN ID RPARAN
          { HMRef $3 }

HeightmapList:
        | Heightmap
          { [$1] }
        | Heightmap HeightmapList
          { $1::$2 }

Volumemap:
        | NOISE LPARAN Num COMMA Num RPARAN
          { Noise3D($3,$3,$3,$5) }
        | NOISE LPARAN Num COMMA Num COMMA
          Num COMMA Num RPARAN
          { Noise3D($3,$5,$7,$9) }
        | ADD LPARAN VolumemapList RPARAN
          { Add3D $3 }
        | OFFSET Position Volumemap
          { Offset3D($2,$3) }
        | VOLUMEMAP LPARAN ID RPARAN
          { VMRef $3 }

VolumemapList:
        | Volumemap
          { [$1] }
        | Volumemap VolumemapList
          { $1::$2 }

%%
```

C.3 F# code

In this section are the source code created in the project.

C.3.1 GameDefinition.fs

```
module Game.GameDefinition


type Result =
    | Won
    | Lost
    | Tie

type Outcome<'state, 'player> = 'state -> 'player ->
    Result option

type Rule<'state, 'player> =
    | PerPlayerRule of ('state -> 'player -> 'state)
    | AllPlayerRule of ('state -> ('player list) -> '
        state)
    | StateRule of ('state -> 'state)

type Game<'state, 'player> = 'player list * (Rule<'state,
     'player> list) * Outcome<'state, 'player>


let EvaluateRule players state rule =
    match rule with
    | PerPlayerRule f ->
        List.fold f state players
    | AllPlayerRule f ->
        f state players
    | StateRule f ->
        f state

let GameStep ((players, rules, outcome) : Game<_,_>)
    state =
    let GameNotEnded player = outcome state player = None
    let activeplayers = List.filter GameNotEnded players
    let EvalRule state rule = EvaluateRule activeplayers
        state rule
    match activeplayers with
    | [] -> (state, false)
    | _ -> (List.fold EvalRule state rules, true)
```

C.3.2 PlayerController.fs

```
module Game.PlayerController


type InputButton =
    | Key of UnityEngine.KeyCode
    | MouseButton of int

type Input =
    | Button of string * InputButton
    | Axis of string * string

type PlayerController = Input list

let PlayerButton keyf mousef controller name =
    let buttonPressed (input : Input) : bool =
        match input with
        | Button(name', inputbutton) when name' = name ->
            match inputbutton with
            | Key(kc) ->
                keyf kc
            | MouseButton(n) ->
                mousef n
        | _ ->
            false
    List.exists buttonPressed controller

let PlayerButtonPressed =
    PlayerButton (UnityEngine.Input.GetKey)
        (UnityEngine.Input.GetMouseButton)
let PlayerButtonDown =
    PlayerButton (UnityEngine.Input.GetKeyDown)
        (UnityEngine.Input.GetMouseButtonDown)
let PlayerButtonUp =
    PlayerButton (UnityEngine.Input.GetKeyUp)
        (UnityEngine.Input.GetMouseButtonUp)

let PlayerAxis controller name =
    let findaxis input =
        match input with
        | Axis (name', _) ->
            name' = name
        | _ -> false
```

```
    match List.tryFind findaxis controller with
    | Some(input) ->
        match input with
        | Axis (_,name') ->
            UnityEngine.Input.GetAxis(name')
        | _ ->
            0.0f
    | _ ->
        0.0f
```

C.3.3 Base.fs

```
module Game.Base


let PositiveMod num m =
    let v = num % m
    if v < 0 then v + m else v

let rec ListFoldMap f state l =
    match l with
    | [] -> ([],state)
    | hd::tl ->
        let (hd', state1) = f hd state
        let (tl', state2) = ListFoldMap f state1 tl
        (hd'::tl', state2)

let ListSplit n l =
    let rec ListSplit' n l1 l2 =
        match (n,l2) with
        | (0,_) -> (List.rev l1,l2)
        | (_,[]) -> (List.rev l1,[])
        | (_,hd::tl) -> ListSplit' (n-1) (hd::l1) tl
    ListSplit' n [] l
```

C.3.4 Position.fs

```
module Game.Voxel.Position

```

```
type Position = float32 * float32 * float32

type Direction =
    | XPos
    | XNeg
    | YPos
    | YNeg
    | ZPos
    | ZNeg
    member this.toString () =
        match this with
        | XPos -> "X+"
        | XNeg -> "X-"
        | YPos -> "Y+"
        | YNeg -> "Y-"
        | ZPos -> "Z+"
        | ZNeg -> "Z-"

let PositionBinop f (x,y,z) (x',y',z') = (f x x', f y y', f z z')

let PositionUnop f (x,y,z) = (f x, f y, f z)

let Distance ((x,y,z) : Position) ((x',y',z') : Position) =
    let sq a = a * a
    sqrt(sq(x+x') + sq(y+y') + sq(z+z'))

let PositionToVector ((x,y,z) : Position) =
    new UnityEngine.Vector3(x,y,z)
let VectorToPosition (v : UnityEngine.Vector3) : Position =
    (v.x, v.y, v.z)

let IncreasePositionInDirection1 dir (x,y,z) =
    match dir with
    | XPos
    | XNeg -> (x,       y+1.0f, z       )
    | YPos
    | YNeg -> (x+1.0f, y,       z       )
    | ZPos
    | ZNeg -> (x+1.0f, y,       z       )

```

```
let IncreasePositionInDirection2 dir (x,y,z) =
    match dir with
    | XPos
    | XNeg -> (x,       y,       z+1.0f)
    | YPos
    | YNeg -> (x      , y,       z+1.0f)
    | ZPos
    | ZNeg -> (x,       y+1.0f,  z     )
```

C.3.5 Mesh.fs

```
module Game.Voxel.Mesh


type 'a Mesh = (UnityEngine.Vector3 list) * (int list) * ('a)


let rec internal AddMesh compareOp ((vs, ts, voxel) as mesh : 'a Mesh) (meshes : 'a Mesh list) : 'a Mesh list =
    match meshes with
    | [] -> [mesh]
    | (vs2, ts2, voxel2)::t when compareOp(voxel, voxel2) = true ->
        (vs2 @ vs, ts2 @ (List.map (fun t -> t + List.length vs2) ts), voxel)::t
    | h::t -> h::(AddMesh compareOp mesh t)

let CombineMeshes compareOp meshlist1 meshlist2 : 'a Mesh list =
    List.fold (fun meshes mesh -> AddMesh compareOp mesh meshes) meshlist1 meshlist2
```

C.3.6 Voxel.fs

```
module Game.Voxel.Voxel

open Game.Base
```

```
open Game.Voxel.Mesh
open Game.Voxel.Position
open System.Text.RegularExpressions

// Voxel: ISL?RRRRGGGGBBBB
// I: Indexed value
//     if 0, the remaining 15 bits determine a simple colored block/air
//     if 1, the remaining 15 bits determine an indexed block
// S: Is this a solid block? Only relevant for I = 0
// L: Does this block emit light? Only relevant for I = 0 & S = 1, i.e. solid blocks
// RRRRGGGGBBBB: 4 bits for each of red/green/blue

(*

Types of voxels needed:
- Solid cube (has color and maybe 'health')
- Light-emitting solid cube (has color and maybe 'health') (color also determines intensity)
- Baked lighting (has color) (color also determines intensity) (normal air included in this)
- Liquid (has color and level)
- Light-emitting liquid (has color and level) (color also determines intensity)
- Indexed value (maybe has 6 bits reserved for 'connected to')

*)

type Voxel = uint16 // "us" affix, for instance 16us

let internal VoxelToRgb (voxel : Voxel) =
    (voxel >>> 8 &&& 15us, voxel >>> 4 &&& 15us, voxel &&& 15us)

let internal VoxelToRgbFloat voxel =
    let ColorToFloat c =
        (float32 c) / 15.0f
    VoxelToRgb voxel |> fun (r,g,b) -> (ColorToFloat r, ColorToFloat g, ColorToFloat b)

```

```
let internal RgbToVoxel (r : uint16, g : uint16, b : uint16) : Voxel =
    let (r', g', b') = (r &&& 15us, g &&& 15us, b &&& 15us)
    (r' <<< 8) + (g' <<< 4) + b'

let internal FloatToColor f =
    f * 15.0f |> round |> uint16

let internal MakeSolid v =
    v ||| (1us <<< 14)

let VoxelIsSolid v =
    v &&& (1us <<< 14) <> 0us

let ColorToVoxel (c : UnityEngine.Color) =
    let FloatToColor f = f * 15.0f |> round |> uint16
    RgbToVoxel (FloatToColor c.r, FloatToColor  c.g, FloatToColor c.b) |> MakeSolid

let VoxelToColor (v : Voxel) =
    let (r,g,b) = VoxelToRgbFloat v
    new UnityEngine.Color(r,g,b)

let airVoxel = RgbToVoxel (8us, 8us, 8us)

let VoxelFromString (s : string) : Voxel =
    let CharToUint16 c =
        if '0' <= c && c <= '9'
        then (uint16 c) - (uint16 '0')
        else if 'a' <= c && c <= 'f'
        then (uint16 c) + 10us - (uint16 'a')
        else if 'A' <= c && c <= 'F'
        then (uint16 c) + 10us - (uint16 'A')
        else 0us
    if Regex.IsMatch(s, "^#[0-9a-fA-F]{3,3}$")
    then
        let r = CharToUint16 s.[1]
        let g = CharToUint16 s.[2]
        let b = CharToUint16 s.[3]
        printfn "Creating voxel from string: %s -> (%d,%d,%d)" s r g b
        RgbToVoxel (r,g,b) |> MakeSolid
    else
```

```
        failwith (sprintf "Given string does not define a
            voxel. Must be '#' followed by 6 hex, i.e.
           '#009d2F'.\nWas given: %s" s)

let VoxelToString (v : Voxel) : string =
    let (r,g,b) = VoxelToRgb v
    let numToHex n =
        match n with
        | 10us -> "A"
        | 11us -> "B"
        | 12us -> "C"
        | 13us -> "D"
        | 14us -> "E"
        | 15us -> "F"
        | _ ->
            if 0us <= n && n <= 9us
            then string n
            else "."
    let colorToString c = (numToHex (c % 16us))
    "#" + (colorToString r) + (colorToString g) + (
        colorToString b)

(*

                 v111
                //|\\
              //  |  \\
            //    |   \\
          //      |    \\
        //        |     \\
  v011 ||\   v101/\    /|| v110
       ||\\    /   \// ||
       || \\ /    //\   ||
       ||  \\   //   \ ||
       ||/  \\//      \||
  v001 \\    ||v010   //   v100
        \\   ||      //
         \\  ||    //
          \\ || //
           \\|//
           v000

            y
            |
```

```
        z    |        x
        \    |       /
         \   |     /
          \  |   /
           \|/

*)

let internal v000 : Position = ( -0.5f, -0.5f, -0.5f)
let internal v001 : Position = ( -0.5f, -0.5f,  0.5f)
let internal v010 : Position = ( -0.5f,  0.5f, -0.5f)
let internal v011 : Position = ( -0.5f,  0.5f,  0.5f)
let internal v100 : Position = (  0.5f, -0.5f, -0.5f)
let internal v101 : Position = (  0.5f, -0.5f,  0.5f)
let internal v110 : Position = (  0.5f,  0.5f, -0.5f)
let internal v111 : Position = (  0.5f,  0.5f,  0.5f)

let VoxelToMesh dir (pos : Position) voxel : Voxel Mesh =
    let offsetVertex (v : Position) =
        PositionBinop (+) pos v |> PositionToVector
    match dir with
    | XPos ->
        ([v100; v101; v110; v111] |> List.map
            offsetVertex, [0;2;3;0;3;1], voxel)
    | XNeg ->
        ([v000; v001; v010; v011] |> List.map
            offsetVertex, [0;1;3;0;3;2], voxel)
    | YPos ->
        ([v010; v011; v110; v111] |> List.map
            offsetVertex, [0;1;3;0;3;2], voxel)
    | YNeg ->
        ([v000; v001; v100; v101] |> List.map
            offsetVertex, [0;2;3;0;3;1], voxel)
    | ZPos ->
        ([v001; v011; v101; v111] |> List.map
            offsetVertex, [0;2;3;0;3;1], voxel)
    | ZNeg ->
        ([v000; v010; v100; v110] |> List.map
            offsetVertex, [0;1;3;0;3;2], voxel)
```

C.3.7 ProceduralGenerator.fs

```
module Game.Voxel.ProceduralGenerator

open Game.Base
open Game.Voxel.Position
open Game.Voxel.Voxel


type Heightmap =
    | Noise2D of float32 * float32 * float32 // hor-scale * ver-scale
    | Plane of float32
    | Add2D of Heightmap list
    | Offset2D of Position * Heightmap
    | HMRef of string // Reference to another height-map

type Volumemap =
    | Noise3D of float32 * float32 * float32 * float32 // x,y,z scale, weight
    | Add3D of Volumemap list
    | Offset3D of Position * Volumemap
    | VMRef of string

type LandscapeDef =
    | Heightmap of Heightmap * LandscapeDef * LandscapeDef
    | AreaMap of Heightmap * (Range list) * LandscapeDef
    | VolumeMap of Volumemap * (Range list) * LandscapeDef
    | LandscapeRef of string
    | Gameobject of string * string
    | VoxelVal of Voxel
    | VoxelRef of string
and Range = float32 * float32 * LandscapeDef

type 'a LandscapeResult =
    | VoxelValue of Voxel
    | Object of 'a

type 'a LandscapeEnv =
    {
    GetHeightmap : string -> Heightmap option;
    GetVolumemap : string -> Volumemap option
    GetLandscape : string -> LandscapeDef option;
```

```
    GetVoxel : string -> Voxel option;
    GetObject : string -> string -> 'a;
    }

let internal NoiseGen =
    new Graphics.Tools.Noise.Primitive.SimplexPerlin()

let rec EvaluateHeightmap (landscapeenv : _ LandscapeEnv) x z hm =
    match hm with
    | Noise2D(sx,sz,weight) when
        sx <> 0.0f && sz <> 0.0f && weight <> 0.0f ->
        NoiseGen.GetValue(x / sx, z / sz) * weight
    | Plane v -> v
    | Add2D(hms) ->
        List.map (EvaluateHeightmap landscapeenv x z) hms |> List.fold (fun a b -> a + b) 0.0f
    | Offset2D((x',y',z'),hm) ->
        (EvaluateHeightmap landscapeenv (x+x') (z+z') hm) + y'
    | HMRef(s) ->
        match (landscapeenv.GetHeightmap s) with
        | None -> 0.0f // ERROR
        | Some hm -> EvaluateHeightmap landscapeenv x z hm
    | _ -> 0.0f // ERROR

let rec EvaluateVolumemap (landscapeenv : _ LandscapeEnv) ((x,y,z) as pos) vm =
    match vm with
    | Noise3D (sx,sy,sz,weight) ->
        if sx <> 0.0f && sy <> 0.0f && sz <> 0.0f
        then NoiseGen.GetValue(x / sx, y / sy, z / sz) * weight
        else 0.0f
    | Add3D vms ->
        List.map
            (EvaluateVolumemap landscapeenv pos) vms
            |> List.fold (fun a b -> a + b) 0.0f
    | Offset3D (pos', volumemap') ->
        EvaluateVolumemap landscapeenv (PositionBinop (+) pos pos') volumemap'
    | VMRef name ->
        match landscapeenv.GetVolumemap name with
```

```
            | Some volumemap' -> EvaluateVolumemap
                landscapeenv pos volumemap'
            | _ -> 0.0f

let rec EvaluateLandscape (landscapeenv : _ LandscapeEnv)
     landscape ((x,y,z) as pos) =
     let evalland = EvaluateLandscape landscapeenv
     match landscape with
     | Heightmap (hm, landbelow, landabove) ->
         let hmy = EvaluateHeightmap landscapeenv x z hm
         let landscape' =
             if hmy > y
             then landbelow
             else landabove
         evalland landscape' pos
     | AreaMap (heightmap, ranges, defaultlandscape) ->
         let hmy = EvaluateHeightmap landscapeenv x z
             heightmap
         match List.tryFind (fun (min,max,_) -> min <= hmy
              && hmy <= max) ranges with
         | None -> evalland defaultlandscape pos
         | Some (_,_,landscape') -> evalland landscape'
             pos
     | VolumeMap (volumemap, ranges, defaultlandscape) ->
         let vmv = EvaluateVolumemap landscapeenv pos
             volumemap
         match List.tryFind (fun (min,max,_) -> min <= vmv
              && vmv <= max) ranges with
         | None -> evalland defaultlandscape pos
         | Some (_,_,landscape') -> evalland landscape'
             pos
     | LandscapeRef (name) ->
         match landscapeenv.GetLandscape name with
         | Some landscape' -> evalland landscape' pos
         | _ -> VoxelValue airVoxel
     | Gameobject (typename, name) ->
         match landscapeenv.GetObject typename name with
         | Some o -> Object o
         | _ -> VoxelValue airVoxel
     | VoxelVal (voxel) ->
         VoxelValue voxel
     | VoxelRef (name) ->
         match landscapeenv.GetVoxel name with
         | Some voxel -> VoxelValue voxel
```

```
        | _ -> VoxelValue airVoxel
```

C.3.8 Chunk.fs

```
module Game.Voxel.Chunk

open Game.Base
open Game.Voxel.Position
open Game.Voxel.Mesh
open Game.Voxel.Voxel


type ChunkData = Voxel [,,]
type Chunk = ChunkData * Position



(*let CreateChunk chunkSize (voxelGenerator : Position ->
    Voxel) chunkCoordinate : Chunk =
    let coord = positionFactor chunkSize chunkCoordinate
    let i = chunkSize |> int
    let data =
        fun x y z -> voxelGenerator (positionAdd coord (
            float32 x, float32 y, float32 z))
        |> Array3D.init<Voxel> i i i
    (data, coord)
*)

let ChunkdataFromString width height depth chartovoxel (s
    : string) : ChunkData =
    let chunkdata = Array3D.create width height depth
        airVoxel
    let chars = s.ToCharArray()
    Array.fold (fun i c ->
        match chartovoxel c with
        | Some voxel when i < width * height * depth ->
            chunkdata.[i % width, (i / width) % height, (
                i / (width * height)) % depth] <- voxel
            i + 1
        | _ -> i
        ) 0 chars |> ignore
    chunkdata
```

```
let ChunkPositionFromVoxelPosition chunkdims : Position -> Position =
    PositionUnop float32 chunkdims
    |> PositionBinop (fun a b -> b / a |> floor)

// Given dimensions of a chunk-array and a chunk-coordinate, this function will calculate
// the position in the chunk-array of the given position
let ChunkArrayPosition chunkAmount (pos : Position) : Position =
    PositionUnop (fun a -> floor a |> int) pos
    |> PositionBinop (fun a b -> PositiveMod b a) chunkAmount
    |> PositionUnop float32

let WorldDim chunksize chunkamount : Position =
    PositionBinop ( * ) chunksize chunkamount |> PositionUnop float32

let LandscapeOffset chunksize chunkamount centerpos =
    centerpos
    |> PositionBinop (+) (WorldDim chunksize chunkamount |> PositionUnop (fun a -> a * -0.5f))
    |> ChunkPositionFromVoxelPosition chunksize
    |> PositionBinop ( * ) (PositionUnop float32 chunksize)

let internal WithinChunk (x,y,z) (chunkdata : ChunkData) =
    x >= 0.0f && y >= 0.0f && z >= 0.0f
    && int x < chunkdata.GetLength(0)
    && int y < chunkdata.GetLength(1)
    && int z < chunkdata.GetLength(2)

let internal Neighbour dir (chunkdata : ChunkData) (pos : Position) =
    let offset =
        match dir with
        | XPos -> (1,0,0)
        | XNeg -> (-1,0,0)
        | YPos -> (0,1,0)
        | YNeg -> (0,-1,0)
        | ZPos -> (0,0,1)
```

```
            | ZNeg -> (0,0,-1)
        let neighborpos = PositionUnop float32 offset |>
            PositionBinop (+) pos
        if WithinChunk neighborpos chunkdata
        then
            let (x,y,z) = PositionUnop int neighborpos
            Some (chunkdata.[x,y,z])
        else None

let internal ShouldDraw voxel neighbour =
        match neighbour with
        | None ->
            VoxelIsSolid voxel
        | Some neighbourVoxel ->
            not (VoxelIsSolid neighbourVoxel) && VoxelIsSolid
                voxel

let rec updateCreatedMeshArray inc1 inc2 (
    meshAlreadyCreated : _ [,,]) pos1 pos2 pos12 ((x,y,z)
    as pos) =
        meshAlreadyCreated.[int x, int y, int z] <- true
        if pos = pos12
        then ()
        elif pos = pos2
        then updateCreatedMeshArray inc1 inc2
            meshAlreadyCreated (inc1 pos1) (inc1 pos2) pos12 (
            inc1 pos1)
        else updateCreatedMeshArray inc1 inc2
            meshAlreadyCreated pos1 pos2 pos12 (inc2 pos)

let rec allGood inc dir voxel (chunkdata : ChunkData) (
    meshAlreadyCreated : _ [,,]) ((x,y,z) as pos1 :
    Position) ((x2,y2,z2) as pos2 : Position) =
        if pos1 <> pos2 && x >= x2 && y >= y2 && z >= z2
        then
            true
        else
            let thisvoxel = chunkdata.[int x, int y, int z]
            let neighbourvoxel = Neighbour dir chunkdata pos1
            let solidneighbour =
                match neighbourvoxel with
                | Some v when (VoxelIsSolid v) ->
                    true
                | _ ->
```

```
                false
        if (ShouldDraw thisvoxel neighbourvoxel && thisvoxel = voxel && not (meshAlreadyCreated.[int x, int y, int z])) || solidneighbour
        then allGood inc dir voxel chunkdata meshAlreadyCreated (inc pos1) pos2
        else false

let createGreedyMeshOnLayer inc1 inc2 dir (chunkdata : ChunkData) meshAlreadyCreated ((x,y,z) as pos) =
    let voxel = chunkdata.[int x, int y, int z]
    let rec greedyMesh pos1 pos2 pos12 caninc1 caninc2 turn1 =
        (*
            On plane:

            (x,y,z)  X-------X  pos1
                     |       |
                     |       |
                     |       |       -> Direction 1
                     |       |
                     |       |
               pos2  X-------X  pos12

                         |
                         v
                     Direction 2

        *)
        match caninc1,caninc2,turn1 with
        | false,false,_ ->
            (pos1,pos2,pos12)
        | true,true,true
        | true,false,_ ->
            // Increment coordinates in direction 1
            let p1 = inc1 pos1
            let p12 = inc1 pos12
            if WithinChunk p12 chunkdata && allGood inc2 dir voxel chunkdata meshAlreadyCreated p1 p12
            then greedyMesh p1 pos2 p12 caninc1 caninc2 (not turn1)
            else greedyMesh pos1 pos2 pos12 false caninc2 (not turn1)
```

```
        | true,true,false
        | false,true,_ ->
            // Increment coordinates in direction 2
            let p2 = inc2 pos2
            let p12 = inc2 pos12
            if WithinChunk p12 chunkdata && allGood inc1 dir voxel chunkdata meshAlreadyCreated p2 p12
            then greedyMesh pos1 p2 p12 caninc1 caninc2 (not turn1)
            else greedyMesh pos1 pos2 pos12 caninc1 false (not turn1)
    greedyMesh pos pos pos true true true

let rec createMeshes inc1 inc2 dir (chunkdata : ChunkData) (meshAlreadyCreated : _ [,,]) ((x,y,z) as pos : Position) meshes =
    if int x >= chunkdata.GetLength(0)
    then createMeshes inc1 inc2 dir chunkdata meshAlreadyCreated (0.0f, y+1.0f, z) meshes
    elif int y >= chunkdata.GetLength(1)
    then createMeshes inc1 inc2 dir chunkdata meshAlreadyCreated (x, 0.0f, z+1.0f) meshes
    elif int z >= chunkdata.GetLength(2)
    then meshes
    elif meshAlreadyCreated.[int x, int y, int z] || not (ShouldDraw chunkdata.[int x, int y, int z] (Neighbour dir chunkdata pos))
    then createMeshes inc1 inc2 dir chunkdata meshAlreadyCreated (x+1.0f, y, z) meshes
    else
        let voxel = chunkdata.[int x, int y, int z]
        let (pos1, pos2, pos12) = createGreedyMeshOnLayer inc1 inc2 dir chunkdata meshAlreadyCreated (x,y,z)
        let (vertices,triangles,_) = VoxelToMesh dir (0.0f,0.0f,0.0f) voxel
        let addVector3Pos (v : UnityEngine.Vector3, (x,y,z) : Position) =
            new UnityEngine.Vector3(v.x + float32 x, v.y + float32 y, v.z + float32 z)
        let newvertices =
            List.zip vertices [(x,y,z);pos2;pos1;pos12]
            |> List.map addVector3Pos
```

```
            let ((x1,y1,z1),(x12,y12,z12)) = (pos1,pos12)
            updateCreatedMeshArray inc1 inc2 meshAlreadyCreated (x,y,z) pos2 pos12 (x,y,z)
            createMeshes inc1 inc2 dir chunkdata meshAlreadyCreated (x+1.0f, y, z) (AddMesh (fun (a,b) -> a = b) (newvertices,triangles,voxel) meshes)

let ChunkToMesh dir ((chunkdata,_) : Chunk) : Voxel Mesh list =
    let (xdim, ydim, zdim) = (chunkdata.GetLength(0), chunkdata.GetLength(1), chunkdata.GetLength(2))
    let meshAlreadyCreated = Array3D.create xdim ydim zdim false
    let inc1 = IncreasePositionInDirection1 dir
    let inc2 = IncreasePositionInDirection2 dir
    createMeshes inc1 inc2 dir chunkdata meshAlreadyCreated (0.0f, 0.0f, 0.0f) []

let DisplayMeshes (go : UnityEngine.GameObject) (chunk : Chunk) (material : UnityEngine.Material) =
    let meshes =
        [XPos; XNeg; YPos; YNeg; ZPos; ZNeg]
        |> List.map (fun dir -> ChunkToMesh dir chunk)
        |> List.fold (CombineMeshes (fun (v1, v2) -> v1 = v2)) []
    let CreateMesh ((ver, tri, voxel) : Voxel Mesh) =
        let go2 = new UnityEngine.GameObject("Voxel: " + (VoxelToString voxel))
        go2.transform.parent <- go.transform
        go2.transform.localPosition <- new UnityEngine.Vector3(0.0f, 0.0f, 0.0f)
        let meshfilter = go2.AddComponent<UnityEngine.MeshFilter>()
        let meshrenderer = go2.AddComponent<UnityEngine.MeshRenderer>()
        let mesh = meshfilter.mesh
        mesh.vertices <- ver |> List.toArray
        mesh.triangles <- tri |> List.toArray
        mesh.uv <- Array.create (List.length ver) (new UnityEngine.Vector2(0.0f,0.0f))
        mesh.RecalculateBounds()
        mesh.RecalculateNormals()
```

```
        meshrenderer.material <- new UnityEngine.Material
            (material)
        meshrenderer.material.color <- VoxelToColor voxel
    List.iter CreateMesh meshes
```

C.3.9 BehaviorTree.fs

```
module Game.BehaviorTree


//open Game.Base


type UnOp =
    | NumNeg
    | LogicNeg
    | Trim
    | Sqrt
    | Absolute

type BinOp =
    | Eq
    | Less
    | LessEq
    | Greater
    | GreaterEq
    | Plus
    | Subtract
    | Multiply
    | Divide
    | StrContains

type NaryOp =
    | Max
    | Min
    | And
    | Or

type Expr =
    | UnaryOp of UnOp * Expr
    | BinaryOp of Expr * BinOp * Expr
    | NaryOp of NaryOp * (Expr list)
```

```
    | String of string
    | Num of float32
    | Bool of bool
    | Act of ActionExpr
and ActionExpr =
    | ObjectRef of string
    | MemberRef of ActionExpr * ActionExpr
    | InvokeAction of string * (Expr list)

type BehaviorTree =
    | Sequence of BehaviorTree list
    | Selector of BehaviorTree list
    | Parallel of int * int * (BehaviorTree list)
    | Decorator of ActionExpr * BehaviorTree
    | Link of string
    | Condition of Expr
    | Action of ActionExpr

type BehaviorTreeStatus =
    | BTSuccess
    | BTFail
    | BTRunning of BehaviorTreeStatus list

(*
let rec EvalExpr evalactionexpr (e : Expr) context =
    match e with
    | UnaryOp(op,e) ->
        let (exp',context') = EvalExpr evalactionexpr e
            context
        (EvalUnop op exp', context')
    | BinaryOp(e1,op,e2) ->
        let (e1',context1) = EvalExpr evalactionexpr e1
            context
        let (e2',context2) = EvalExpr evalactionexpr e2
            context1
        (EvalBinop op e1' e2', context2)
    | NaryOp(op,es) ->
        let rec foldmap f state l =
            match l with
            | [] -> ([],state)
            | hd::tl ->
                let (hd', state1) = f hd state
                let (tl', state2) = foldmap f state1 tl
                (hd'::tl', state2)
```

```
            let (es',context') = foldmap (EvalExpr evalactionexpr) context es
            (EvalNaryop op es', context')
        | Act a ->
            evalactionexpr a context
        | v -> (v, context)
and EvalUnop op e =
    match (op,e) with
    | (NumNeg, Num n) -> Num (-n)
    | (LogicNeg, Bool b) -> Bool (not b)
    | (Trim, String s) -> String (s.Trim())
    | (Sqrt, Num n) when n >= 0.0f -> Num (sqrt n)
    | (Absolute, Num n) -> Num (abs n)
    | _ -> Bool false // ERROR
and EvalBinop op e1 e2 =
    match (op,e1,e2) with
    | (Eq, String v1, String v2) -> Bool (v1 = v2)
    | (Eq, Num v1, Num v2) -> Bool (v1 = v2)
    | (Eq, Bool v1, Bool v2) -> Bool (v1 = v2)
    | (Less, String v1, String v2) -> Bool (v1 < v2)
    | (Less, Num v1, Num v2) -> Bool (v1 < v2)
    | (LessEq, String v1, String v2) -> Bool (v1 <= v2)
    | (LessEq, Num v1, Num v2) -> Bool (v1 <= v2)
    | (Greater, String v1, String v2) -> Bool (v1 > v2)
    | (Greater, Num v1, Num v2) -> Bool (v1 > v2)
    | (GreaterEq, String v1, String v2) -> Bool (v1 >= v2)
    | (GreaterEq, Num v1, Num v2) -> Bool (v1 >= v2)
    | (Plus, String v1, String v2) -> String (v1 + v2)
    | (Plus, Num v1, Num v2) -> Num (v1 + v2)
    | (Subtract, Num v1, Num v2) -> Num (v1 - v2)
    | (Multiply, Num v1, Num v2) -> Num (v1 * v2)
    | (Divide, Num v1, Num v2) when v2 <> 0.0f -> Num (v1 / v2)
    | (StrContains, String v1, String v2) -> Bool (v1.Contains v2)
    | _ -> Bool false // ERROR
and EvalNaryop op els =
    match els with
    | [e] -> e
    | e1::tl ->
        let e2 = EvalNaryop op tl
        match (op,e1,e2) with
```

```
        | (Max, Num v1, Num v2) -> Num (if v1 > v2 then v1 else v2)
        | (Min, Num v1, Num v2) -> Num (if v1 < v2 then v1 else v2)
        | (And, Bool v1, Bool v2) -> Bool (v1 && v2)
        | (Or, Bool v1, Bool v2) -> Bool (v1 || v2)
        | _ -> Bool false // ERROR
    | _ -> Bool false // Empty list

// Need: Status for previous calculations, string -> BehaviorTree, ActionExpr -> BTResult, Expr -> bool
let rec EvaluateBehaviorTree evala evalae evald fb (context : 'a) (bt : BehaviorTree) (bts : BehaviorTreeStatus) : (BehaviorTreeStatus * 'a) =
    let evalrec = EvaluateBehaviorTree evala evalae evald fb
    match bt with
    | Sequence (btlist) ->
        match (bts,btlist,BTInitial) with
        | (BTInitial,[],_) ->
            (BTSuccess,context)
        | (BTRunning (SerialNode(0,bts')),hd::tl,_)
        | (BTInitial,hd::tl,bts') ->
            match evalrec context hd bts' with
            | (BTSuccess,context') -> evalrec context' (Sequence tl) BTInitial
            | (BTRunning (SerialNode (n,bts2)),context') -> (BTRunning (SerialNode (n+1, bts2)), context')
            | (_,context') -> (BTFail,context')
        | (BTRunning (SerialNode(n,bts')),_::tl,_) when n <= btlist.Length ->
            match evalrec context (Sequence tl) (BTRunning(SerialNode(n-1,bts'))) with
            | (BTRunning(SerialNode(n',bts2)),context') -> (BTRunning(SerialNode(n'+1,bts2)), context')
            | v -> v
        | _ -> (BTFail, context)
    | Selector (btlist) ->
        match (bts,btlist,BTInitial) with
        | (BTInitial,[],_) ->
            (BTFail,context)
        | (BTRunning (SerialNode(0,bts')),hd::tl,_)
```

```
        | (BTInitial,hd::tl,bts') ->
            match evalrec context hd bts' with
            | (BTSuccess,context') -> (BTSuccess,context')
            | (BTFail,context') -> evalrec context' (Selector tl) BTInitial
            | (BTRunning (SerialNode (n,bts2)),context') -> (BTRunning (SerialNode (n+1, bts2)), context')
            | (_,context') -> (BTFail,context')
        | (BTRunning (SerialNode(n,bts')),_::tl,_) when n <= btlist.Length ->
            match evalrec context (Selector tl) (BTRunning(SerialNode(n-1,bts'))) with
            | (BTRunning(SerialNode(n',bts2)),context') -> (BTRunning(SerialNode(n'+1,bts2)), context')
            | v -> v
        | _ -> (BTFail, context)
    | Parallel (0,_,_) -> (BTSuccess,context)
    | Parallel (_,0,_) -> (BTFail,context)
    | Parallel (succ,fail,btlist) ->
        match bts with
        | BTInitial ->
            let btss = BTRunning(ParallelNode(List.replicate (List.length btlist) BTInitial))
            evalrec context bt btss
        | BTRunning(ParallelNode(btss)) when List.length btss = List.length btlist ->
            match (btlist,btss,succ,succ-1,fail,fail-1) with
            | (_::bttl,(BTSuccess as v)::btstl,_,succ,fail,_)
            | (_::bttl,(BTFail as v)::btstl,succ,_,_,fail) ->
                let (bts',context') = evalrec context (Parallel(succ,fail,bttl)) (BTRunning(ParallelNode btstl))
                match bts' with
                | BTRunning(ParallelNode btss) -> (BTRunning(ParallelNode(v::btss)), context')
                | v -> (v,context')
            | (bt::bttl,bts::btstl,_,_,_,_) ->
```

```
                let (bts',context') = evalrec context bt bts
                match bts' with
                | BTSuccess
                | BTFail -> evalrec context' (Parallel(succ,fail,btlist)) (BTRunning(ParallelNode(bts'::btstl)))
                | _ ->
                    match evalrec context' (Parallel(succ,fail,bttl)) (BTRunning(ParallelNode(btstl))) with
                    | (BTRunning(ParallelNode btstl'),context2) -> (BTRunning(ParallelNode(bts'::btstl')),context2)
                    | v -> v
            | _ -> (BTRunning(ParallelNode []),context)
        | _ -> (BTFail,context)
    | Decorator (acexp,bt) ->
        let evalbt context = evalrec context bt
        evald evala evalbt acexp context
    | Link (name) ->
        let bts' =
            match bts with
            | BTRunning (SoloNode bt) -> bt
            | _ -> BTInitial
        match fb name with
        | Some bt -> evalrec context bt bts'
        | _ -> (BTFail,context)
    | Condition (exp) ->
        let (exp', context') = EvalExpr evala exp context
        match exp' with
        | String s when s <> "" -> (BTSuccess, context')
        | Num n when n > 0.0f -> (BTSuccess, context')
        | Bool true -> (BTSuccess, context')
        | _ -> (BTFail, context')
    | Action (acexp) ->
        evalae evala acexp context
*)
```

C.3.10 Prefab.fs

```
module Game.Prefab


open Game.Base
open Game.Voxel.Position
open Game.Voxel.Voxel
open Game.Voxel.ProceduralGenerator
open Game.BehaviorTree


type PrimitiveValue =
    | PrimInt of int
    | PrimFloat of float32
    | PrimStr of string
    | PrimBool of bool
    | PrimPosition of float32 * float32 * float32

type LevelDef =
    {
    Landscape : string;
    PlayerSpawnPoint : Position;
    Attr : Map<string, PrimitiveValue>;
    }

type ItemDef =
    {
    Appearance : Map<string, PrimitiveValue>;
    Attr : Map<string, PrimitiveValue>;
    }

type NpcDef =
    {
    Appearance : Map<string, PrimitiveValue>;
    Attr : Map<string, PrimitiveValue>;
    BehaviorTree : string;
    }

type PlayerDef =
    {
    Appearance : Map<string, PrimitiveValue>;
    Attr : Map<string, PrimitiveValue>;
    BehaviorTree : string;
    Camera : Map<string, PrimitiveValue>;
```

```
    Inventory : Map<string, string>;
    }

type BTType =
    | BTPlayer
    | BTNpc

type GameDef =
    {
    Levels : Map<string, LevelDef>;
    Player : PlayerDef;
    Items : Map<string, ItemDef>;
    Npcs : Map<string, NpcDef>;

    BehaviorTrees : Map<string, BehaviorTree>;

    Voxels : Map<string, Voxel>;
    Heightmaps : Map<string, Heightmap>;
    Volumemaps : Map<string, Volumemap>;
    Landscapes : Map<string, LandscapeDef>;
    }

let DefaultPlayer =
    {
    Appearance = Map.empty;
    Attr = Map.empty;
    BehaviorTree = "";
    Camera = Map.empty;
    Inventory = Map.empty;
    }

let DefaultGame =
    {
    Levels = Map.empty;
    Items = Map.empty;
    Npcs = Map.empty;
    Player = DefaultPlayer;
    BehaviorTrees = Map.empty;
    Heightmaps = Map.empty;
    Volumemaps = Map.empty;
    Landscapes = Map.empty;
    Voxels = Map.empty;
    }
```

C.3.11 State.fs

```
module Game.State


open System.Collections.Generic
open Game.Base
open Game.Voxel.Position
open Game.Voxel.Voxel
open Game.Voxel.ProceduralGenerator
open Game.Voxel.Chunk
open Game.BehaviorTree
open Game.Prefab

type VisualVoxelObject = UnityEngine.GameObject * Chunk *
    float32

type Item =
    {
    appearence : VisualVoxelObject;
    attr : Map<string, PrimitiveValue>;
    }

type Npc =
    {
    appearence : VisualVoxelObject;
    attr : Map<string, PrimitiveValue>;
    behaviorTree : BehaviorTree;
    behaviorTreeStatus : BehaviorTreeStatus;
    }

type Player =
    {
    appearence : VisualVoxelObject;
    attr : Map<string, PrimitiveValue>;
    behaviorTree : BehaviorTree;
    behaviorTreeStatus : BehaviorTreeStatus;
    inventory : Map<string, Item>;
    }

type ItemNpc =
    | ItemRes of ItemDef
    | NpcRes of NpcDef

```

```
type Landscape =
    {
    chunks : (VisualVoxelObject option)[,,];
    landscapeOffset : Position;
    refreshChunks : Position list;
    voxelCreator : Position -> ItemNpc LandscapeResult;
    chunkDimensions : int * int * int;
    chunkAmount : int * int * int;
    }

type State =
    {
    landscape : Landscape;
    items : Item list;
    npcs : Npc list;
    player : Player;
    attr : Map<string, PrimitiveValue>;
    }


let CreateVisualObject (appearence : Map<_,_>) goTitle :
    VisualVoxelObject =
    let scale =
        match appearence.TryFind "Scale" with
        | Some (PrimFloat f) when f > 0.0f -> f
        | Some (PrimInt i) when i > 0 -> float32 i
        | _ -> 1.0f
    let chunkstr =
        match appearence.TryFind "Chunk" with
        | Some (PrimStr s) -> s
        | _ -> ""
    let voxelcolor =
        match appearence.TryFind "Color" with
        | Some (PrimStr s) -> s
        | _ -> "#0ff"
    let chartovoxel c =
        match c with
        | '1' -> Some (VoxelFromString voxelcolor)
        | '0' -> Some airVoxel
        | _ -> None
    let getdim s =
        match appearence.TryFind s with
        | Some (PrimInt i) when i > 1-> i
        | _ -> 1
```

```
    let (w,h,d) =
        (getdim "Width", getdim "Height", getdim "Depth")
    let chunkdata =
        ChunkdataFromString w h d chartovoxel chunkstr
    let centerpos : Position =
        match appearence.TryFind "Center" with
        | Some (PrimPosition (x,y,z)) -> (x,y,z)
        | _ ->
            (float32 w / 2.0f,
                float32 h / 2.0f,
                float32 d / 2.0f)
    (new UnityEngine.GameObject(goTitle),
        (chunkdata, centerpos),
        scale)

let ShowVisualObject ((go,chunk,scale) : VisualVoxelObject) (p : Position) material parent =
    // Remove all children of go
    while go.transform.childCount > 0 do
        UnityEngine.Object.Destroy(go.transform.GetChild(0).gameObject)
    DisplayMeshes go chunk material
    go.transform.localScale <- new UnityEngine.Vector3(scale,scale,scale)
    match parent with
    | None -> go.transform.position <- PositionToVector p
    | Some transform ->
        go.transform.parent <- transform
        go.transform.localPosition <- PositionToVector p
    ()

let SetupCharacterController ((go,(chunkdata,_),scale) as visob : VisualVoxelObject) =
    let cc =
        match go.GetComponent<UnityEngine.CharacterController>() with
        | null -> go.AddComponent<UnityEngine.CharacterController>()
        | c -> c
    cc.slopeLimit <- 90.0f
    cc.stepOffset <- 1.1f
    cc.height <- (float32 (chunkdata.GetLength(1))) * scale
```

```
    cc.radius <- (float32 (chunkdata.GetLength(0) +
        chunkdata.GetLength(2))) * scale / 4.0f
    visob

let CreateItem (itemdef : ItemDef) : Item =
    {
    appearence = CreateVisualObject itemdef.Appearance "
        Item";
    attr = itemdef.Attr;
    }

let CreateNpc fb (npcdef : NpcDef) : Npc option =
    match fb npcdef.BehaviorTree with
    | Some bt ->
        Some {
        appearence = CreateVisualObject npcdef.Appearance
            "Npc";
        attr = npcdef.Attr;
        behaviorTree = bt;
        behaviorTreeStatus = BTRunning [];
        }
    | _ -> None

let CreatePlayer fb fi (playerdef : PlayerDef) playerpos
    : Player option =
    let getitem name =
        match fi name with
        | None -> None
        | Some item -> Some (CreateItem item)
    let optionalAdd f map key value =
        match f value with
        | Some v -> Map.add key v map
        | _ -> map
    let optionFilter f =
        Map.fold (optionalAdd f) Map.empty
    match fb playerdef.BehaviorTree with
        | Some bt ->
            let (go,_,_) as visob = CreateVisualObject
                playerdef.Appearance "Player"// |>
                SetupCharacterController
            go.transform.Translate (PositionToVector
                playerpos)
            Some {
            appearence = visob;
```

```
                attr = playerdef.Attr;
                behaviorTree = bt;
                behaviorTreeStatus = BTRunning [];
                inventory = playerdef.Inventory |> optionFilter getitem
                }
            | _ -> None

let internal ChunkCreator landscapeenv landscape ((_,y,_) as pos) =
    match landscape with
    | None when y >= 0.0f -> VoxelValue airVoxel
    | None -> VoxelFromString "#530" |> VoxelValue
    | Some landscape' ->
        EvaluateLandscape landscapeenv landscape' pos

let CreateInitialState (gamedef : GameDef) chunksize ((w,h,d) as chunkamount) levelname : State option =
    match gamedef.Levels.TryFind levelname with
    | Some level ->
        let player = gamedef.Player
        let fo t =
            match t with
            | "Item" ->
                fun s ->
                    match gamedef.Items.TryFind s with
                    | None -> None
                    | Some itemdef -> Some (ItemRes itemdef)
            | "Npc" ->
                fun s ->
                    match gamedef.Npcs.TryFind s with
                    | None -> None
                    | Some npcdef -> Some (NpcRes npcdef)
            | _ -> fun _ -> None
        let startpos = level.PlayerSpawnPoint
        let landscapeOffset = LandscapeOffset chunksize chunkamount startpos
        let landscapeenv =
            {
            GetHeightmap = gamedef.Heightmaps.TryFind;
            GetVolumemap = gamedef.Volumemaps.TryFind;
            GetLandscape = gamedef.Landscapes.TryFind;
            GetVoxel = gamedef.Voxels.TryFind;
```

```
            GetObject = fo;
            }
        let landscape =
            {
            chunks = Array3D.create w h d None;
            landscapeOffset = landscapeOffset |>
                PositionBinop (+) (PositionUnop (fun a ->
                a * 5.0f) (WorldDim chunksize chunkamount)
                );
            refreshChunks = [];
            voxelCreator = ChunkCreator landscapeenv (
                gamedef.Landscapes.TryFind level.Landscape
                );
            chunkDimensions = chunksize;
            chunkAmount = chunkamount;
            }
        match CreatePlayer gamedef.BehaviorTrees.TryFind
            gamedef.Items.TryFind player startpos with
        | None -> None
        | Some p ->
            let state =
                {
                landscape = landscape;
                npcs = []; // Initially empty, will be
                    created with the landscape
                player = p;
                items = []; // Initially empty, will be
                    created with the landscape
                attr = level.Attr;
                }
            Some state
    | _ -> None
```

C.3.12 CreateGame.fs

```
module Game.CreateGame


open UnityEngine
open Game.GameDefinition
open Game.PlayerController
open Game.Base
```

```
open Game.Voxel.Position
open Game.Voxel.Voxel
open Game.Voxel.ProceduralGenerator
open Game.Voxel.Chunk
open Game.BehaviorTree
open Game.State
open Game.Prefab
open Game.BTGparser
open Game.BTGlexer


let internal readbtg s =
    let lexbuf = Lexing.LexBuffer<_>.FromString s
    try
        Game.BTGparser.GameDefinition BTGlexer.tokenize lexbuf
    with e ->
        let pos = lexbuf.EndPos
        let line = pos.Line + 1
        let column = pos.Column
        let message = e.Message
        let lastToken = new System.String(lexbuf.Lexeme)
        Debug.Log (sprintf "Parse failed at line %d, column %d:" line column)
        Debug.Log (sprintf "Last token: %s" lastToken)
        failwith "Parse error"
let internal readfile f =
    System.IO.File.ReadAllText(f) |> readbtg


// Rules to be created:
// PlayerRule that executes players' behavior trees
(*let MovePlayer =
    fun (state : State) (pc : PlayerController) ->
        let player = state.player
        let (bts,(state',(player',_))) = evalBt (state,(player,pc)) player.behaviorTree player.behaviorTreeStatus
        {state' with player = {player' with behaviorTreeStatus = bts}}

// StateRule that executes npcs' behavior trees
let MoveNpcs =
    fun (state : State) ->
```

```
        let MoveNpc (state : State) name =
            match state.npcs.TryFind name with
            | None -> state
            | Some npc ->
                let (bts,(state',npc')) = evalBt (state, npc) npc.behaviorTree npc.behaviorTreeStatus
                {state' with npcs = state'.npcs.Add (name,npc')}
        let npcnames =
            Map.toList state.npcs
            |> List.map (fun (s,_) -> s)
        List.fold MoveNpc state npcnames
*)

// StateRule that centers and updates the view of player
let CenterView =
    fun (state : State) ->
        let (go,_,_) = state.player.appearence
        let newcenter = VectorToPosition go.transform.position
        let landsize = WorldDim state.landscape.chunkDimensions state.landscape.chunkAmount
        let oldcenter = PositionBinop (+) state.landscape.landscapeOffset (PositionUnop (fun a -> a * 0.5f) landsize)
        // If player is a certain distance from the center of the view:
        let (dx,dy,dz) = PositionBinop (fun a b -> a - b |> abs) newcenter oldcenter
        let (dw,dh,dd) = state.landscape.chunkDimensions |> PositionUnop float32 |> PositionUnop (fun a -> a * 2.0f)
        if dx > dw || dy > dh || dz > dd
        then
            let getchunkpos = ChunkPositionFromVoxelPosition state.landscape.chunkDimensions
            let getmax = PositionBinop (+) (PositionUnop float32 state.landscape.chunkAmount)
            let oldmin = state.landscape.landscapeOffset |> getchunkpos
            let oldmax = getmax oldmin
```

```
            let newlandoffset = LandscapeOffset state.landscape.chunkDimensions state.landscape.chunkAmount newcenter
            let newmin = newlandoffset |> getchunkpos
            let newmax = getmax newmin
            let (x,y,z) = PositionUnop int newmin
            let (w,h,d) = state.landscape.chunkAmount
            Debug.Log(sprintf "Recentering\nlandoffset: %O\nnewlandoffset: %O\nnewcenter: %O\nlandsize: %O\noldcenter: %O\nold min and max: %O - %O\nnew min and max: %O - %O" (state.landscape.landscapeOffset) newlandoffset newcenter landsize oldcenter oldmin oldmax newmin newmax)
            let Within (xmin,ymin,zmin) (xmax,ymax,zmax) (x,y,z) =
                xmin <= x && x <= xmax && ymin <= y && y <= ymax && zmin <= z && z <= zmax
            // Calculate the positions in the new view, but not in the old, and add these to the list of chunks to be updated
            let newchunks =
                [
                for i in x..x+w-1 do
                for j in y..y+h-1 do
                for k in z..z+d-1 do
                let pos = (float32 i, float32 j, float32 k)
                if (Within newmin newmax pos) && not (Within oldmin oldmax pos)
                then yield pos
                ]
            // Filter positions that are no longer in the view
            let oldupdates = List.filter (Within newmin newmax) state.landscape.refreshChunks
            // Sort the list of positions by distance to the player
            let updates =
                newchunks @ oldupdates
                |> List.sortBy (Distance newcenter)
            {state with landscape = {state.landscape with refreshChunks = updates; landscapeOffset = newlandoffset}}
```

```
        else
            state

let CreateChunk (gamedef : GameDef) material (state : State) (chunkpos : Position) =
    let (w,h,d) = state.landscape.chunkDimensions
    let pos0 =
        PositionUnop float32 state.landscape.chunkDimensions
        |> PositionBinop ( * ) chunkpos
    let (chunkx,chunky,chunkz) =
        ChunkArrayPosition state.landscape.chunkAmount chunkpos
        |> PositionUnop int
    let (go,chunk,_) =
        match state.landscape.chunks.[chunkx,chunky,chunkz] with
        | None ->
            let visob = (new GameObject(sprintf "Chunk %O" chunkpos),(Array3D.create w h d airVoxel,pos0),1.0f)
            state.landscape.chunks.[chunkx,chunky,chunkz] <- Some visob
            visob
        | Some visob -> visob
    let (chunkdata,_) = chunk
    let chunkpositions =
        [
        for i in 0..(w-1) do
        for j in 0..(h-1) do
        for k in 0..(d-1) do
        yield (i,j,k)
        ]
    let createVoxel voxelcreator state (x,y,z) =
        let pos =
            PositionUnop float32 (x,y,z)
            |> PositionBinop (+) pos0
        let v = state.landscape.voxelCreator pos
        //Debug.Log (sprintf "Voxel @ %O -> %O" pos v)
        match v with
            | Object (ItemRes itemdef) ->
                let item = CreateItem itemdef
                // Show item
```

```
                ShowVisualObject item.appearence pos material None
                // Add collider to item
                let (go,(chunkdata,_),scale) = item.appearence
                let size =
                    (0,1,2)
                    |> PositionUnop chunkdata.GetLength
                    |> PositionUnop (fun a -> float32 a * scale)
                let collider = go.AddComponent<BoxCollider>()
                collider.size <- PositionToVector size
                go.AddComponent<Rigidbody>() |> ignore
                // Add item to state
                {state with items = item::state.items} // TODO - name of item
            | Object (NpcRes npcdef) ->
                match CreateNpc gamedef.BehaviorTrees.TryFind npcdef with
                | None -> state
                | Some npc ->
                    // Show npc
                    ShowVisualObject npc.appearence pos material None
                    // Set up character controller on npc
                    SetupCharacterController npc.appearence |> ignore
                    // Add npc to state
                    {state with npcs = npc::state.npcs} // TODO - name of npc
            | VoxelValue v ->
                chunkdata.[x,y,z] <- v
                state
    let state' = List.fold (createVoxel state.landscape.voxelCreator) state chunkpositions
    ShowVisualObject (go, chunk, 1.0f) pos0 material None
    // Add mesh collider to all childs of go
    if go.transform.childCount > 0
    then
        for i in 0..(go.transform.childCount)-1 do
            go.transform.GetChild(i).gameObject.AddComponent<MeshCollider>() |> ignore
    state'
```

```

let DrawLandscape (gamedef : GameDef) material state =
    let (updatenow, updatelater) = ListSplit 4 state.landscape.refreshChunks
    let state' = List.fold (CreateChunk gamedef material) state updatenow
    {state' with landscape = {state'.landscape with refreshChunks = updatelater}}

let OutcomeFunction : Outcome<State, PlayerController> =
    fun (state : State) _ ->
        match state.player.behaviorTreeStatus with
        | BTSuccess -> Some Won
        | BTFail -> Some Lost
        | _ -> None

let DefaultController : PlayerController =
    [
    Button("Jump", Key(KeyCode.Space));
    Button("Jump", MouseButton(1));
    Button("Attack", Key(KeyCode.LeftShift));
    Button("Attack", MouseButton(0));
    Button("Reload", Key(KeyCode.R));
    Button("Ranged Weapon", Key(KeyCode.Alpha2));
    Button("Melee Weapon", Key(KeyCode.Alpha1));
    Axis("Forward", "Vertical");
    Axis("Strafe", "Horizontal");
    Axis("Look", "Mouse Y");
    Axis("Turn", "Mouse X")
    ]

let CreateGameFromFile material chunksize chunkamount filename levelname : (Game<State, PlayerController> * State) option =
    try
        let gamedef = readfile filename
        let initState = CreateInitialState gamedef chunksize chunkamount levelname
        match initState with
        | Some state ->
            let players = [DefaultController] // Not actually used
            let rules = [StateRule CenterView; StateRule (DrawLandscape gamedef material); (*
```

```
                PerPlayerRule MovePlayer; StateRule MoveNpcs;*)]
            Some ((players, rules, OutcomeFunction), state)
        | _ ->
            None
    with e ->
        Debug.Log "Error creating game"
        None
```

C.3.13 CreateGameClass.fs

```
namespace Game

open UnityEngine
open Game.GameDefinition
open Game.PlayerController
open Game.State
open Game.CreateGame

type CreateGameClass = class
    inherit MonoBehaviour

    val mutable public GameFile : string
    val mutable public LevelName : string
    val mutable public ChunkSize : int
    val mutable public ChunkAmountHor : int
    val mutable public ChunkAmountVer : int
    val mutable public DefaultMaterial : Material

    val mutable private GameState : State option
    val mutable private Game : Game<State, PlayerController> option

    member this.Start () =
        let size = (this.ChunkSize, this.ChunkSize, this.ChunkSize)
        let amount = (this.ChunkAmountHor, this.ChunkAmountVer, this.ChunkAmountHor)
        Debug.Log "Creating game..."
        match CreateGameFromFile this.DefaultMaterial size amount this.GameFile this.LevelName with
```

```
            | Some (game, state) ->
                Debug.Log "Game created"
                this.GameState <- Some state
                this.Game <- Some game
            | _ ->
                Debug.Log "Was unable to create game"
                this.GameState <- None
                this.Game <- None

    member this.Update () =
        match (this.GameState, this.Game) with
        | (Some state, Some game) ->
            let (state',_) = GameStep game state
            this.GameState <- Some state'
        | _ ->
            ()

end
```

Bibliography

[1] Jesper Juul, *The Game, the Player, the World: Looking for a Heart of Gameness*, In Level Up: Digital Games Research Conference Proceedings, edited by Marinka Copier and Joost Raessens, 30-45. Utrecht University, 2003, http://www.jesperjuul.net/text/gameplayerworld/

[2] Katie Salen & Eric Zimmerman, *Rules of Play - Game Design Fundamentals*, MIT Press, Cambridge, 2003

[3] Mojang AB, *Minecraft*, video game, http://www.minecraft.net

[4] Zachtronics Industries, *Infiniminer*, video game, http://thesiteformerlyknownas.zachtronicsindustries.com/?p=713

[5] William E. Lorensen, Harvey E. Cline, *Marching Cubes: A high resolution 3D surface construction algorithm*, Computer Graphics, Vol. 21, Nr. 4, July 1987

[6] The Transvoxel™ Algorithm, http://www.terathon.com/voxels/

[7] Eric Lengyel, *Transition Cells for Dynamic Multiresolution Marching Cubes* Journal of Graphics, GPU, and Game Tools. Vol. 15, No. 2 (2010), A K Peters. DOI: 10.1080/2151237X.2011.563682

[8] Ken Perlin, *Making Noise*, Slides from GDCHardcore, 1999, http://www.noisemachine.com/talk1/

[9] Ken Perlin, *Noise hardware*, Real-Time Shading SIGGRAPH Course Notes (2001), Olano M., (Ed.).

[10] Bjoern Knafla, *Introduction to Behavior Trees*, http://www.altdevblogaday.com/2011/02/24/introduction-to-behavior-trees/

[11] Maxis, *Spore*, 2008, video game, http://www.spore.com/

[12] Chris Hecker, *My Liner Notes for Spore/Spore Behavior Tree Docs*, http://chrishecker.com/My_Liner_Notes_for_Spore/Spore_Behavior_Tree_Docs

[13] Unity Technologies, *Unity*, game development platform, http://unity3d.com/

[14] Scott D Roth, *Ray Casting for Modeling Solids*, Computer Graphics and Image Processing 18 (2), February 1982