AV Engine Documentation Documentation

Release 0.1

Edward Herbert

Jan 23, 2022

Contents:

Welcome to the documentation for the AV Engine.

The AV engine is a game engine written in C++ that I've been writing in my free time, as part of a larger project. The eventual goal and purpose of the engine is to support an as of yet unreleased game. This documentation aims to serve as a means for me to document how everything works and fits together, as well as how to use the functions of the engine itself.

Features of the engine include:

- Implementation of a streamable open world
- Real-time physics
- A heavy focus on scripting and data driven content
- Abstraction of platform specifics
- An Entity Component System
- Support for serialisation
- A powerful dialog system

As well as this I've tried to architect the engine as cleanly as possible, both as a learning experience and to increase its future maintainability. The engine tries to be as data driven as it can, while still trying to be as optimised as possible.

The engine is intended to support a game with the following features:

- 3D graphics
- A streamable open world
- Explorative style RPG
- Top down style gameplay where the camera follows the player
- Extensibility via scripts and mods
- Cross Platform

However, it can be used to create other sorts of projects as well. For instance, 2d games are completely possible using the engine's 2d features. Furthermore, with its data driven nature, many different kinds of projects can be created using this engine.

Much of the engine is based on external libraries. These include:

- Ogre3D
- Bullet Physics
- EntityX
- Squirrel (Scripting Language)
- SDL for desktop windowing
- A few smaller utility libraries

CHAPTER 1

Architecture

1.1 Architecture Overview

This section of the documentation will focus on explaining the architecture of the engine. I'll give an overview of the majour components of the engine, as well as try and explain the design decisions I made and why they were made.

The AV engine has been designed with a focus on simplicity and maintainability, as well as being fit for a specific use case. This is not supposed to be a general purpose game engine, and therefore I've made some decisions about what to include and what not to include. Many of these decisions are intended to support this idea of simplicity, and maintainability. Bear in mind that this is intended to be a project completed by a single person, and a lot of design decisions have been based around me and my limited resources.

1.1.1 Bird's eye view

The engine is split into a number of components.

1.1.2 Outline of Components

Much of the engine is based around an event messaging system, which is supplied by the event manager. The purpose of this is to help with the decoupling of a number of the majour components. Rather than components talking directly to the target component, they instead favour the use of the event system, which will broadcast the event to components which have subscribed.

The largest set of components in the engine is the world, which contains the actual logic of the game world. Inside the world are a variety of other systems which work together to provide the streamable open world present in the gameplay. The world itself is rather complex, and involves the entity system, physics system, level streaming and also provides support for deserialisation. The world can be shutdown and restarted during the runtime of the game, as a new world save file is requested to be loaded.

Outside of the world, the other components focus on things that aren't directly coupled with the world. These components rarely communicate with one another directly and instead use the event system. Components such as the Input component deals with abstracting the details of inputs to game specific actions. So rather than space key pressed, the

input system deals with jump_input_sent. A similar thing could be said for the window component, which acts as a system abstraction for output. For instance, it contains an SDL window for desktop environments, and could be extended to support others implementations for output on other platforms.

Specifics of the majour engine components are discussed in more detail in their own sections.

1.2 World

The world is a large set of components and systems in the engine. It acts as a base in which gameplay elements can occur, for instance, physics, entities and 3d objects. To make a game with the engine, the user does not necessarily have to create a world, however many common requirements for game development are provided by the world.

The world is a single class that deals with the encapsulation of a range of other components. These components include but are not limited to:

- Physics
- Entities
- Streamable content such as chunks

Only a single world can exist at a time.

The world supports implementation of save states (serialisation). When creating a world, the user is able to supply a save file with which the world should construct itself. The save file will be used to bring the world to a state identical to when the save file was originally created.

When the world is created, it will initialise all the other components which are involved in the world and begin loading content from there.

1.2.1 Creation and lifetime

A world can be constructed using the script function:

```
_world.createWorld();
```

It can be destroyed using the script function:

```
_world.destroyWorld();
```

The user is given the choice of when (if at all) to construct the world. Rather than creating a single world at the startup of engine, a better solution is to allow the user to specify when they wanted the world created. Consider for example a menu system that's presented to the player on startup. This menu would show a title screen, then allow them to select a save. Up until they've selected the save there would be no need for the world, and now that a save has been selected the world can be constructed with that save in mind. Now remember that the game will most likely want to have a 'quit to menu' option. In this case, the world would be destroyed to make room for another. The user might go to the menu screen and select another save, and the process would continue as before.

The world is therefore required to be flexible in its creation and construction, as it could happen an unlimited amount of times during runtime. In order to create a new world, the old one must first be destroyed.

1.2.2 Access

Some script functions are reliant on the world for their operation. If the `_world.createWorld()` function hasn't been called by the time these functions are called, an error will be thrown.

For instance:

```
//No world exists at this point.
local e = _entity.create();
//calling this function here will throw an error.
```

The script api documentation describes in more detail which functions are reliant on the existence of the world.

1.3 Entity Manager

Note: The following is a re-design of the old entity system.

It is not fully implemented yet!

The Entity Manager is the part of the engine responsible for managing entities in the world. The entity system is based on an EntityComponentSystem, which is provided by the library EntityX. Quite simply, an Entity Component System is an architectural approach to dealing with entities. It favours composition of entities rather than a strict and ridgid entity hierarchy. Entities are tied together by a list of components which represent simple aspects of the entity. Entities are a combination of components, which can be combined to create more complex behaviour. Components do nothing but store data, and the components themselves are manipulated by systems, which provide the functionality.

The Entity Manager itself ties together a number of parts of the engine. It aims to be the governing body over entities, and plays a vital role in their creation, destruction and modification. Outside of the entity part of the engine, the fact that the entity component system is implemented by entity x is unknown. I aim to have no includes from entity x outside of the entity system. This is intentional, as it means I can fully lock down the functionality of the system.

1.3.1 Functions of the Entity Manager

The Entity Manager itself performs a number of jobs.

These jobs include:

- Creating and destroying entities, and performing the necessary bookeeping
- Positioning and moving entities
- Coordinating the operation of the systems.

There are a number of functions which are delegated off to other parts of the system, such as managing the entity chunks. From the perspective of the user working outside the entity orientated part of the engine, there is no mention of entity x. This helps to prevent interference from parts of the engine which shouldn't be interfering. For example, in the prototype of this system, I did much of my work by passing around entity handles. However with a handle you have complete access to the entity and its components. With this I could have destroyed the position component, or directly altered it without letting the entity manager know. In this style of working, there were no solid guarantees as to how things operated.

In this new system, there is no direct access to the entities or their components. Everything is passed through designated c++ functions which serve to facilitate the manipulation of components.

Creating Entities

Entities are created with a single component by default. This is the position component, which is a simple component which contains a slot position representing where the entity resides in the world. All entities in the system have a position, and the user doesn't have to worry about creating it either, as that is handled by the engine on entity creation.

There is no direct access to the position component, instead the user must use the supplied `setPosition()` function in the entity manager. This will take care of all the necessary bookeeping when moving an entity (such as moving meshes to reflect the new position, and so on).

Iterating all entities with a position component is an effective way to iterate all entities in the world.

Component Manipulation

Component manipulation is an important part of the entity process. Components contain plain data, but this data needs to be read and written to. As I don't directly expose these components to the user, how does this work?

Instead, I have a series of static functions implemented in c++ which deal with altering the components. If a health component had an int health value, I would have a static c++ class which contains functions to get and set the health value. This process would be repeated for each component the engine offers. In this way I don't offer direct access to the components, but still allow them to be read and modified.

The disadvantage of this approach is the complexity. There is an increased ammount of code that needs to be written for each component. Furthermore the lack of direct access could be potentially seen as a performance issue, as external functions need to be called to do simple things.

However, there are some majour benefits to this approach. The first and most substantial is the ability to entirely police the modifications to the components. As mentioned previously, some components are there for the benefit of the engine, rather than the user, for instance the position component. In the old system there was nothing to stop the user from modifying the values in this component to whatever they wanted. However, the engine often wanted to do some vital bookeeping whenever an entity was moved (moving meshes, moving physics shapes). Because of this direct access I wasn't able to guarantee that the positions in the component matched everything else.

Consider another example, which would be the health component as mentioned before. It contains a single health value, but surely if this value was less than or equal to 0 the entity should be destroyed? That was how I wanted the system to work, but how would it know this? As before I was just editing the value in the entity. If I wanted to check if the entity had recently died, I would have to either include the checks where the value was set, or have a system that checks the value each frame. Both are untidy solutions. However, with an exposed c++ function I can do all the checks there, as there will only ever be one place that sets the value of the component.

This is also a majour benefit for the scripting system, as now it has a single place to call to do simple manipulations. Previously the scripts would have had to contain their own checks per exposed function, which would have been a duplication of code.

The Player Component

The player component is an important component for the engine. The engine itself has no solid definition of what the player should be, or how it should act, but the position component does help it keep track of which entity represents the player. In and of itself the position component is an empty component which does nothing. It is mearly used by the engine to designate the player.

There can only be one entity at a time with the player component. The user has no direct access to the component, except for the `setPlayer()` function in the entity manager. This will give the player component to that entity, and remove it from any other entity which had it until then.

1.3.2 Entity Tracking

Entity Tracking is the method I use to manage entity lifetime. Say for instance an entity was created in the world, when would I want it to be destroyed? As the engine supports an open world game, that would most likely be when the player goes far enough away from that entity. It is important to make sure that entities are managed, as in some

cases the engine can add them to the world automatically. It would make sense therefore that they can also be removed automatically.

That is the solution that entity tracking provides. Entities are sorted into entity chunks. These chunks are very similar to slots in the Slot Manager, in that they represent sections of the world. An Entity Chunk is essentially a section of the map, that contains a list of which entities reside within it. The player has a load radius, and if they move far enough away from that chunk, it is destroyed.

Chunks are created lazily, meaning that if there are no entities in that part of the map a chunk won't be created. If an entity is inserted into an area of the map that has no entities, a chunk will be created to contain it. If the final entity in a chunk is removed, the chunk and its list won't be destroyed until it goes out of range. If the map switches, all tracked entities are destroyed.

Upon entity creation, the entity manager asks the user whether they want this entity to be tracked or not. Essentually this decision boils down to, 'Do you want me to deal with the entity's deletion, or do you want to be responsible for it?' If the entity is not tracked by the engine, it will not be deleted automatically. That becomes the user's job.

The engine preferes to manage entities itself though. Entities created as part of a chunk creation are tracked by default.

The EntityManager exposes an api to track and untrack entities. The squirrel scripts also have access to an api to track and untrack entities by an eId. Really, the only time an entity should be untracked is when they're being used for something specific in a script, such as a cutscene.

If an entity is untracked, it will not disappear until something manually destroys it. Furthermore, it will persist engine serialisation, meaning you might be stuck with it forever.

So please make sure that if you come to untrack an entity it is eventually destroyed.

1.3.3 Tracking and Untracking

Entities can be either tracked or untracked on the fly. This is useful if you decide you want to delegate control of the entity back to the engine or visa versa.

Code like this will create an entity tracked, and then untrack it.

```
local e = _entity.createTracked(SlotPosition()); //Create a tracked entity.
_entity.untrack(e); //The entity is now untracked.
_entity.track(e); //And now it's re-tracked.
```

1.3.4 Destruction of tracked entities

An entity being tracked means the emphasis is put on the engine to manage its lifetime. This means that the engine has control over when it is destroyed. As such, the user needs to be aware during their interaction with tracked entities that they might be destroyed while using them. This destruction is not random, and will most of the time happen when the player goes far enough away from the entity, or the entity makes some sort of movement. As such, the user needs to be aware that tracked entities should involve more checks in the scripts than their untracked counterparts.

The engine exposes a way to check if an entity is still valid.

```
local e = _entity.createTracked(SlotPosition());
if(e.valid()){
    print("Doing some stuff with a valid entity.");
}
```

This method works for both tracked and untracked entities.

The best practice for this situation would be to avoid direct scripted interaction with tracked entities as much as possible. If heavy scripting is involved for an entity, untracked entities should be used instead.

1.3.5 Entity Callbacks

The entity manager exposes an interface to run squirrel scripts on event occurance. These take the form of entity callbacks. Ultimately, this is based on the callback script system, where closures (functions) can be entered into a script file and executed on demand. This system allows the user extended control of entity operation.

An example of a callback script would be:

```
function moved(entity){
    _entity.destroy(entity);
}

function destroyed(entity){
    print("destroyed");
}
```

Callback scripts are attached to entities like this:

```
_component.script.add(e, _settings.getDataDirectory() + "/EntityScript.nut");
```

An entity can only have one callback script at a time, and they are attached to the entity as components. Similarly to other components, the script component can be removed, and a different one put in its place.

In the example you can see the layout of a callback script. Functions are entered and executed based on their names. For instance, the 'moved' function would be executed whenever the specific entity moves. The designated names are set, meaning that you must specify a function called 'moved' if you want to receive movement callbacks.

Entity callbacks also take an eid as a parameter. This allows the user to perform functionality based on the specific entity that has caused the event. In the example above you can see that whenever an entity moves, it is immediately destroyed. This will lead to the destroyed function callback being executed before the moved callback returns. Although much of this functionality is quite useless, it demonstrates how more complex functionality could be implemented. Callback functions have access to the complete functionality of the scripting system, and as such are very powerful.

Implementation details

As mentioned previously, the entity callback system is based on the regular script callback system (the same that handles scripted states). However, there are some further changes in functionality to improve efficiency. The most prominent is that entity callback scripts are reference counted. If multiple entities use the same script, it is only loaded in memory once. The only change between executions is the eid that's passed to the script.

This means that something like

```
function moved(entity){
    x <- 10;
}
```

will persist between all entities that use this script.

When no entities reference this script, it will be unloaded.

It is also worth mentioning that for efficiency's sake, the user should try and only define the callbacks that they actually need. For instance, defining the `moved` callback is actually quite an expensive operation, as then the moved callback will be called each time that entity moves. These calls can become expensive, so try and avoid them if possible.

Callback scripts are scanned for entries at load, which is more efficient than each time a callback is fired.

1.4 Physics

The engine provides support for physics features, provided by Bullet physics. This allows the implementation of both dynamic, realtime physics and collision detection. The engine's implementation of Bullet is provided in a threaded manner, meaning good performance should be possible.

Much fo the emphasis for the physics implementation has been shifted to C++, meaning Squirrel scripts should aim to setup their physics simulation desires and let the engine fulfil them. For instance, the collision system allows the user to specify a number of flags to determine what counts as a collision, meaning less logic has to be performed by Squirrel. This allows the workload to be shifted onto the physics thread, and better performance to be achieved as a whole.

The engine provides two types of physics worlds to the user:

- Dynamics world
- Collision world

1.4.1 Dynamics World

The dynamics world implements a realtime dynamic physics simulation. In this sense, bodies are inserted into a world, and interact with one another dynamically, similar to how they would in the real world. The user is able to create bodies with specified shapes, and attach them to meshes or entities. The attached objects will be positioned as the rigid body moves throughout the world. Scripts have the ability to create rigid bodies, and assign them according to their needs.

1.4.2 Collision World

The engine implements bullet collision as collision worlds. Collision worlds have no concept of dynamic movement. They care only about which objects are intersecting one another. The user is able to have multiple worlds at a single time, depending on their needs for collision. A maximum of 4 worlds is allowed, and they are specified at startup in the setup file. These worlds employ a system of sender and receiver objects, which allow events to be passed to scripts as required. Similarly to the dynamics world, a number of of methods are used to allow the user to specify exactly what they wish to be notified of, meaning much of the work can be shifted back to the engine and C++ code.

1.4.3 Script Details

There is a number of technical details regarding the script api which should be kept in mind.

Dynamics Overview

The following example shows how to create a rigid body, insert it into the world, and cause it to move an entity or mesh around.

```
//Create an entity
local en = _entity.create(SlotPosition());

//Create a cube shape with diameters 1x1x1.
local shape = _physics.getCubeShape(1, 1, 1);
//Create a rigid body using this shape.
local body = _physics.dynamics.createRigidBody(shape);
_physics.dynamics.addBody(body);
```

(continues on next page)

(continued from previous page)

```
//Assign the rigid body to the entity by providing it as a component.
_component.rigidBody.add(en, body);
//The user can also assign to a mesh.
::mesh <- _mesh.create("cube");
mesh.attachRigidBody(body);
//NOTE a rigid body can only be attached to one object at a time.
```

Collision Overview

Creating objects in the collision world can be performed with the following snippet.

```
function emptyCallback(){
    print("I'm called on collision!");
}

//Tables are used to contain construction info, and passed to the construction␣
↪functions.
local senderTable = {
    "func" : emptyCallback
    "id" : 1, //An arbuitrary id to be assigned to the object.
    "type" : _COLLISION_PLAYER, //The type of object which this is.
    "event" : _COLLISION_INSIDE //The events it should send upon.
};
local receiverInfo = {
    "type" : _COLLISION_PLAYER, //For a collision to occur, this needs to match up␣
↪with the type of the sender.
};

//Create a shape the same as the physics world.
local cubeShape = _physics.getCubeShape(1, 1, 1);

//When calling the collision world, the world is referenced by this id in the␣
↪function.
local receiver = _physics.collision[0].createReceiver(receiverInfo, cubeShape);
local sender = _physics.collision[0].createSender(senderTable, cubeShape);

//Attach the objects to the world.
_physics.collision[0].addObject(sender);
_physics.collision[0].addObject(receiver);
```

Collision senders and receivers

The collision world uses the concept of senders and receivers to perform collisions. Only if a sender and receiver collide can an event occur.

As an example, if the user wanted to build a damage dealing projectile system, this could be easily fulfilled with the collision world. The user might create a sender which sends fire damage. It has no need to move, but sits there and waits for a receiver, attached to the player, to touch it. When the callback occurs, the script can apply the damage to the object.

Or, if creating a trigger system, the user might want a function to run when the player gets close enough to a door. In this case a sphere sender would be placed by the door, waiting for the player receiver to approach it.

Scripting objects are reference counted

This means that when script objects such as shapes, rigidBodies and collision sender and receivers run out of references, they are destroyed. The intention of this is to help avoid memory leaks, and make destruction of objects as simple as possible.

Collision objects which are attached to entities or meshes will gain a reference, and can be obtained later. However, objects which are not attached and lost all references will be destroyed.

The following snippet demonstrates this:

```
//If never used for anything, this shape would be destroyed once this variable loses␣
↪scope.
local shape = _physics.getCubeShape(1, 1, 1);

//A body has been created which references the shape.
local body = _physics.dynamics.createRigidBody(shape);
//Previously the shape would have been destroyed when set to something else. Now it␣
↪survives while the body survives.
shape = 0;

//A mesh is created and the body is attached to it. The body's lifespan extends until␣
↪the mesh is destroyed.
::mesh <- _mesh.create("cube");
mesh.attachRigidBody(body);
//Resetting the body will not destroy it as it has these references.
body = 0;
//The body now has no references. Both it and its shape are destroyed. If the shape␣
↪was used by other bodies it would survive.
mesh.detachRigidBody();
```

The user must specify how many collision worlds they wish to use

Information about the collision worlds is specified in the setup file. If the user does not specify any of this information, the engine assumes the user does not wish to use any collision worlds.

Collision world object properties

The engine provides the user with access to a number of object properties which can be set during collision object creation. Once set they cannot be changed.

An example is shown below.

```
function emptyCallback(){
    print("I'm called on collision!");
}

//Tables are used to contain construction info, and passed to the construction␣
↪functions.
local senderTable = {
    "func" : emptyCallback,
    "id" : 1, //An arbuitrary id to be assigned to the object.
    "type" : _COLLISION_PLAYER | _COLLISION_ENEMY | _COLLISION_OBJECT,
    "event" : _COLLISION_INSIDE | _COLLISION_ENTER | _COLLISION_LEAVE
};
local receiverInfo = {
```

(continues on next page)

(continued from previous page)

```
    "type" : _COLLISION_PLAYER | _COLLISION_ENEMY //For a collision to occur, this␣
↪needs to match up with the type of the sender.
};

local receiver = _physics.collision[0].createReceiver(receiverInfo, _physics.
↪getCubeShape(1, 1, 1));
local sender = _physics.collision[0].createSender(senderTable, _physics.
↪getCubeShape(1, 1, 1));
_physics.collision[0].addObject(sender);
_physics.collision[0].addObject(receiver);
```

These parameters are provided as part of the table input. The possible parameters are:

Key	Value
type	The type that this sender or receiver is. This is used to refine collisions. This input is a bit mask, so an object can have multiple types.
func	Either a squirrel closure or a string representing a function to be called when a collision occurs. If a string is provided, the "path" field is also expected to be populated.
path	A res path to a squirrel script. If populated, the engine will use this path to determine which script to call, expecting a function with the same name as previously provided in the "func" parameter.
id	A numeric id, provided by the user. This id is not used by the engine, and is purely a means to identify an object.

When specifying a receiver object, only the type value is used.

Reduce the number of callbacks used

Callback scripts can be loaded once and shared between multiple objects. For best performance, limit the number of different scripts to as few as possible.

As an example, the user might be using a collision world to perform trigger events. The most efficient way to meet this requirement is this:

```
function playerEnter(id){
    switch(id){
        case 0:
            print("Player reached door");
            break;
        case 1:
            print("Player reached gate");
            break;
    }
}

function playerLeave(id){
    print("do something");
}
```

In this example, the same function is used, and the sender id is used to determine how to handle the event. This approach is much more memory efficient than assigning a unique function for each sender.

The user might assign a different script for the colliders per chunk of the game. If colliders share the same closure and script, callback script and closure information can be shared in memory. Generally the user should prefer to specify a script and function rather than a callback, as it is better structured for larger projects.

1.4.4 Threading details

Physics in the engine is implemented using threads. The user should be aware of this fact, as there is a chance they will meet issues as a result of the latency between the worker thread and the main thread. Script functions to add or remove bodies, as well as functions such as set body position are subject to a delay between threads. This delay is often only a single frame between a request being made, and it being fulfilled.

Functionality such as getting the position of objects can be performed from the main thread, as a copy is made of useful data such as this.

1.5 Slot Manager

The Slot Manager is the response to the problem of streaming a continuous open world. It allows pieces of level data to be loaded in and out of memory as the player moves around. This level data can include meshes, terrain, physics information and nav mesh data. It also provides a solution to the problem of floating point precision.

A number of problems and issues arise when attempting to stream a large world. These include:

- Memory constraints
- Floating point precision issues
- Loading times

The SlotManager employs the concept of *Slots* and *Chunks* to solve this issue.

A Slot is a coordinate in the world.

A Chunk is a piece of data that fits into a slot.

In order to stream an open world, you need to split it into smaller individual *chunks* of data. This data is loaded in and presented in the world depending on where the player is at that moment. As the player moves around, new chunks are loaded and inserted, and old ones are deleted. In this sense the game gives the impression that the whole world is loaded at a single time, when in fact it is only partly loaded.

1.5.1 Floating Point Precision

Floating point precision is a fundamental problem in computer science, in which computers have trouble representing floating point (decimal) numbers of a high magnitude. The bigger the whole number gets, the less of a resolution the decimal number can have.

For a 3d simulation this can have a detrimental effect, and is a classic problem for simulations with a large world. With a lower resolution of decimal value, problems which require precision such as rendering and physics cannot function as intended. Commonly, jumpy renderings become common as the precision at which the computer can represent decimals decreases.

1.5.2 Slots and their benefit

The problem of floating point precision is combated with Slots. Slots would have a set size in world coordinates, say 500 units. That means the origin would have a value of

```
x:0 y:0
```

The position 500 in world coordinates would be:

```
x:1 y:1
```

Because the size of a single slot is 500x500. In between these slots is another measurement, which is represented as an Vector3.

```
x:1 y:0 (100, 0, 100)
```

This vector3 allows you to represent any position within slot space. This coordinate system is known as *SlotPositions*.

The origin of the world is set in the top left corner of the coordinate system. This means that positive x moves in the right direction, and positive z moves downwards from the slot positions.

The intention of this system is to allow an abstraction of the true origin of the world. The engine allows the user to set the origin to any SlotPosition, and then all objects which rely on a regular three float coordinate system are placed relative to this. The engine abstracts SlotPositions to a class which manages these conversions. It contains the ability to retrieve the value of the SlotPosition as a Vector3, which is relative to the user defined origin.

This system means the user can specify a very large world without any concern for floating point precision. Entities, nodes and physics objects can all be positioned as a SlotPosition in a global world coordinate.

1.5.3 Origin Shifting

The engine allows the user to warp the origin at any time they like. This is a process called origin shifting. Here, meshes, physics objects, entities and any other object in the world will be offset by the required value in respect to the new origin.

This process should be completely seamless to the player.

However, origin shifting should be performed sparingly, as it is very expensive. Generally the user will want to perform an origin shift when the player has moved a given distance from the origin, say for instance 2000 units. In order to achieve suitable performance, the user might want to consider techniques such as origin shifting when changing maps, or during a cutscene.

1.5.4 Chunks

Each slot in the world can have data associated with it. As the player moves around, they will want to see new things be loaded in and out of the engine.

Chunks are a type of world data which are inserted into slots. Each chunk has a slot associated with it, which defines when it is loaded, and where its level data appears in the world.

1.5.5 Recipies and Chunks

Recipies are a concept used by the slot manager to construct chunks. A recipe is a collection of plain data structures which describe how to construct a chunk. Recipe data can be re-used as many times as necessary.

Recipies are loaded into memory using a threaded approach. This means the data for the world is loaded in the background without any interruption for the player.

Recipies allow the threading system to load chunks into an intermediate state before their actual construction. This helps prevent the problems posed by blocking IO, as slow file read speeds won't cause the main thread to hang.

When a chunk is requested to be constructed into the world, the recipe will first be loaded into memory before any construction happens. Recipies are stored in memory until they are destroyed to make room for a newer recipe. When a chunk is deconstructed, the recipe won't necessarily be destroyed along with it. The slot manager operates by maintaining a maximum number of recipies that can be loaded at a time. This way, if the chunk is later required to be reconstructed, and the recipe still exists in memory, then it won't have to be read in from the file system again.

1.5.6 Scripts and loading chunks

The Slot Manager has been designed to ask few questions about what it's asked to load. All of the actual logic of what needs to be loaded happens outside of the Slot Manager, with much of this being the ChunkRadiusLoader. This is a class which calculates which chunks need to be loaded based on the radius from the player. The Slot Manager itself simply does as it's told, only performing minimal sanity checks.

Much of the api for the Slot Manager has been intentionally not exposed to scripts. This is because from a scripting point of view I felt the internal workings of the Slot Manager were too low level and complex. For instance, the slot manager has no concept of chunk garbage collection, meaning if a script accidentally loads a chunk and forgets to unload it, it'll remain there for the rest of the engine runtime.

Scripts are able to tell the slot manager to load a recipe, which can be used for pre-loading areas, however there is no direct way to tell the manager to activate or construct chunks. The intended way to control the loading of chunks is to alter the position of the player. The player position is an abstract concept, and coupled with the entity system is easy to change. The whole point of the slot manager is to stream the world as the player moves around it. Components such as the chunk radius checks will react as the player position moves around and manage chunks accordingly. This has a number of advantages, such as removing the possiblilty that scripts can activate a chunk which actually lies outside the floating point safe zone, as the safe zone will have moved as the player position moves. In this process, by moving the player position, scripts can affect active chunks.

- It means scripts don't have to deal with constructing and de-constructing chunks.
- All logic of which scripts need to be loaded can be fed through the radius loader.

Lots of the more complex functionality is going to be implemented in c++ as compared to Squirrel. Things like teleporting the player will be exposed to scripts, but actually implemented in the engine. That means the teleportation code can take advantage of the Slot Manager's functionality with things like pre-creating a chunk. In this way the scripts can use the flexiblity of the engine without having to expose complex system to the user.

1.5.7 Loading Procedure

For much of its api the Slot Manager will perform a number of steps. The Slot Manager uses a threaded approach for loading chunks, and because of this not all requests are completed immediately. For instance, a request to construct a chunk might take a number of update ticks to actually display something on the screen. This is a variable ammount of time depending on how fast the user's computer can load the content from disk.

The Slot Manager takes care of a lot of the heavy lifting for the user. For instance you can call the activateChunk function on a chunk that has never been activated before. The engine will deal with the procedure of loading and construction as well as activation. This means the user doesn't have to make a call to construct chunk, and then check that the chunk is constructed first before calling activating it.

It is also entirely possible to call other api functions on chunks in between their loading cycle. Say for example you requested to construct a chunk, and then during its loading cycle you then made a call to activate that chunk, the engine will switch the operation to perform on the chunk midway, and you will get an activated chunk in the end. This way the user doesn't have to check if the chunk is done loading.

Chunks have a few different states that can be set before their load. These are mostly activate and construct. A constructed chunk isn't necessarily visible, but its meshes and other content has been inserted into the world. An activated chunk is visible, and by its nature has also been constructed. The engine provides a number of api calls to influence how chunks are dealt with.

The recipe system implemented in the SlotManager uses a number of recipe slots. Recipies will be loaded into these slots, when they're requested, and then replaced when the space they occupy is needed. If more recipies are currently pending than there are recipe slots, the request will be pushed into a queue. This queue will be depleated as recipe jobs complete. Once a recipe has finished, and its construction requests are performed, a chunk request will be popped from the queue to take the recipe slot. In this way hundreds of requests more than the size of the recipies list can be requested without any being lost.

The Slot Manager works by providing an output to the user as soon as it's ready and tries to be as simple as possible.

The procedure to construct a chunk is very similar to the above. The only difference is the chunk completion request will be a construction request. The activate request is similar because it also involves constructing the chunk.

1.5.8 Map Switching

The engine allows scripts to specify when to switch maps. This is done like this:

```
_slotManager.setCurrentMap("mapName");
```

Maps are identified by their string name.

Switching maps involves the following procedure:

- Destroys all tracked entities.
- Destroys all chunks (meshes, physics shapes)

1.5.9 Interiors (Not implemented yet)

An interior is a special type of map which does not utilise the streamable world as described above. Their purpose if an optimisation of the map switching that takes place when the player needs to go somewhere different. They act as a pseudo map switch, where the map isn't really switched at all. Instead the chunks of the current map are hidden, and the player is transported to a smaller scene.

The use case for these is if the player wanted to enter a house. If the inside of the house was to be a different map, a lot of background work would need to be done so that the player could visit a relatively small area (destroying entities, destroying geometry, destroying physics shapes). Interiors solve this problem.

So rather than a map switch happening, an interior can be activated which contains a simple scene like structure. There are no chunks or slots within an interior, and everything is loaded up front. This means that there is a limit on how large an interior can be, and they make no effort to address the floating point problem described above. This is because interiors should never be that large.

The c++ and squirrel offers an api to set an interior as active.

1.6 Serialisation

In and of itself, the process of serialising (saving) a game world is a complex procedure. It involves a number of considerations at the engine level, as well as careful planning as to what should be serialised and what shouldn't.

Serialisation in the context of games is being able to save the state of the game to disk at a given point so that it can be recovered to that state on demand. While seeming like a simple operation, there are a number of things that needs to be considered.

- What actually needs to be serialised and what can be left out?
- What procedure does the game need to be aware of when starting itself up?
- How do I make sure nothing gets duplicated (An entity was serialised, but the engine might automatically create another on map load)?
- How do we make sure that the state the game was in when saved is identical to what would be re-created?
- How do we deal with changes in the world data (maps) or the engine (new version supports new things)?

1.6.1 What needs to be serialised?

The absolute minimum possible should be serialised. This is not just to save space on the user's hard disk. Serialisation poses an interesting problem in what the right ammount is.

I could write some code to essentually dump the memory of the engine to disk. On startup I would just load that into memory and carry on from where I was.

However, that would quickly become unmanageable. What if during development something in the game was changed, i.e the contents of a map file? Well if that map got caught up in the serialisation and was just dumped from disk then it wouldn't change.

What if some of the data is pointless to serialise? For instance, I could serialise every shape in the physics world, but what if some of that data was nothing but static shapes defined by the map that will never change? If they don't change what's the point of serialising them. Furthermore, not serialising them solves the problem described above, where if something is changed in the map data, the save would update to reflect that.

Ultimately, the absolute minimum should be serialised.

1.6.2 Startup and duplication

On startup the game needs to be aware of what was previously saved. Otherwise it could run into an issue of things being duplicated. Say for instance that I had serialised a group of entities that exist in part of the map. Something would have had to create them in the first place for them to be serialised. This would most likely have been one of the procedures in the world, like creating entities as part of a chunk load.

The problem then would be that when this map is visited the same group of entities would be created again, which might be a duplication of what was created by loading the serialised data. The solution in that case would be to remember which chunks were loaded at the point of serialisation, and only insert the entities described in the serialised data.

1.6.3 Accounting for edge cases

What if the player saves in a state where an entity moved somewhere strange i.e they moved somewhere as part of a cutscene. You would expect that when the save is loaded, the entity would still be in its strange place.

That makes this an edge case, which can sometimes become difficult to account for.

Methods of Serialisation

As described above, serialisation is a complex problem, and solutions change depending on the needs of the game. In some games it is perfectly acceptable to designate where the player is allowed to save, and this allows the developers more control over what actually gets saved. However if a game wants to offer more flexibility as to when the user can save, the developers have to be aware of more potential issues.

Really there are two sides to the serialisation spectrum, this being serialising everything to be sure, or not serialising anything but the bare essentials. While the two have their benefits, really the best solution is somewhere in the middle.

Serialisation in the engine

The engine is required to support two methods of serialisation, manual saves and auto saves. Realistically, the two try and achieve a similar goal, however they do this in different ways.

An autosave is sort of like a service provided by the engine. Rather than the player having to worry about saving their progress, the engine can be made to do this automatically just to be sure. Autosaves are carried out at specific 'checkpoints' by the user, and would always be done transparently. For instance, during a fast travel, when approaching a momentus part of the story, or after having achieved something like first arrival in a new location.

Manual saves are saves which are performed manually by the user. This would see them going into a menu and asking for the game to be saved. The important thing to remember here is that the user has control of when this happens. You might want to take complete control away from the player i.e they can't save mid way through a boss fight, however this provides a lot more flexiblity than the fixed point nature of the autosaves.

1.6.4 Autosaving

By its nature, autosaves are performed without the request of the player. They exist more for convenience and safety, meaning that even if the player forgets to manually save their game, something will be saved none the less. This puts the autosaving procedure in an interesting place. It needs to happen without the player realising, and it doesn't need to be fully accurate. Let's say for instance that the player has just fast-travelled somewhere. In the process of doing this the engine would have cleared its state anyway, so there's not much state about the world to remember other than 'this is where the player is going to go'. In most cases the checkpoint system would work similarly to this.

The information about the world that can be assumed is assumed, and as little information as information is serialised. The majour benefit of doing this is speed and simplicity. Realistically, the player won't be particularly bothered about the exact position of entities in an autosave, because they didn't make it directly. If the player was being attacked by entities when reaching the checkpoint, these entities would not be serialised. In this case, if the player was to load this save, they wouldn't be exactly as they were before the save was made (i.e not being chased by entities). This is not a problem though, as they didn't physically make the save they most likely won't notice.

This makes the autosaving procedure much simpler, and means that it can be completed very quickly.

1.6.5 Manual saves

Manual saves are a more rigorous form of save. They are requested by the user, and can be performed in a much wider array of places than their counterpart. They essentually take a snapshot of the game as it was when the save happened. In order to achieve this, it does a few things differently to the auto saves.

It still tries to limit what is serialised as much as possible (static meshes, physics shapes), but some things are still serialised. Entities are serialised, and loaded back into the engine in the state they were in before. This includes serialisation of their components and other relevant data.

Given this increased flexibility, the engine has to be careful to avoid the problems described further above.

1.7 Dialog System

The engine contains an implementation of a customisable and flexibile dialog system. It is built with simplicity in mind, as well as user configurability. Ultimately the system is built for very simple dialog, but can also be used for rudimentary scripting.

1.7.1 Design

The dialog system for the avEngine contains two main parts to its implementation, the c++ implementation, and the user implementation.

Dialog scripts themselves are written in xml. The content which needs to be displayed, as well as its order is contained within this file. Dialog scripts make it easy to specify a number of common dialog requirements, such as player choice, variable dialog paths, spoken text per character, so on.

A system to compile these scripts and execute them is implemented in c++. However, in order to keep this truly data driven, a second squirrel implementation is expected to be provided by the user. The engine assumes that the user has their own desires in how dialog should be displayed to the player, and therefore actual implementation of this process is left to them. The engine allows the user to define a script which contains the dialog implementation. This script should implement callback functions, which get called when something in the dialog occurs. Say for instance a new piece of text was spoken. A callback function is simply called passing that string as a parameter. The user's implementation could do anything with that really, from printing it to the screen, or just printing it to the console (or both).

They are able to create any sort of dialog box design or function. Because of this, the uses of the dialog system are quite substantial.

1.7.2 Dialog Scripts

The dialog system is built around dialog scripts. These are scripts which contain the actual dialog. Here is an example.

```
<Dialog_Script>
<b id = "0" sz = "5">
    <ts>The dialog starts here!</ts> <!--This is a blocking tag, meaning the dialog␣
↪will block until it's unblocked by the implementation.-->
    <ts>This is some dialog.</ts> <!--ts means just a direct string. In the future <t>
↪ will mean an id in the localisation system.-->
    <ts>This is some more.</ts>
    <jmp id = "1" /> <!--Switch to dialog block 1. Execution will continue until␣
↪another blocking tag is reached.-->
    <ts>This is some more.</ts> <!--This piece of dialog will never be reached,␣
↪unless the jmp tag was invalid in some way-->
</b>

<b id = "1">
    <ts>Some other bits of dialog.</ts>
    <!--We reach the end of the dialog here, as there's nothing else left to run.-->
</b>
</Dialog_Script>
```

This example shows a few things working together.

Dialog blocks

A section of the code which contains a sequence of 'tags'. Essentially these are bits of dialog. Blocks are executed from the top to the bottom, until the end is reached. Once the end of any block is reached the end of the dialog is reached.

However, different tags do different things, and there are more examples than just printing dialog.

Dialog Tags

Tags are different 'entries' in a dialog block. They're named tags after xml tags. There are a number of different tags, each with a different purpose.

In the example above two tags are seen within a block, `ts` and `jmp`. `ts` is a piece of text. `jmp` is a tag to jump to a different block. So in this example above it would start at block 0, and print out the `ts` entries until it reaches the `jmp` command. It would then jump to block 1 and execute from its beginning. It would keep executing until it reaches the end of the blocks, in which case the dialog finishes.

There are significantly more tags in existence than given in this example, all of which do something different. Their combination allows the user to create all sorts of dialog.

Blocking Tags

Certain tags block until the command to unblock the execution is given. For instance, the system would want to give the user bits of dialog at a time, rather than all at once. That way they would have time to read it. So tags such as `ts` block the execution of the dialog system until the unblock function is called.

1.7.3 Referencing variables

The dialog system has support for reading variable values during execution. This functionality is provided by the value registry system.

Two variable registries are known to the dialog manager. These are global and local registries. Global variables are those contained within the global value registry. Local variables are stored within a value registry owned by the dialog manager. Each time a dialog execution ends, the local registry is cleared.

Setting values in the registry looks like this:

```
_registry.set("playerName", "Wallace"); //Global
_dialogSystem.registry.set("dogName", "Gromit"); //Local
```

The local registry is intended to be populated by scripts before the start of the dialog. For instance, a script might determine where it wants the player to walk to after a certain point. Then it can be passed to the system during execution. Rather than having to call scripts throughout the dialog to determine things, they can just be batched at the start of the dialog. Furthermore, this helps to prevent pollution of the global namespace. If a value is only important to the current dialog, it can be set and forgotten about, knowing it will automatically be deleted when exection is finished.

The user is recommended to prefer local variables over global variables.

Variables are referenced in dialog scripts using either the $$ or ## notifiers, with $ being global and # being local. A variable name is written as, $globalVarName$ or #localVarName#, with the value in between the notifiers being the variable name. Here is an example of its usage:

```
<Dialog_Script>
    <b id = "0">
        <ts>Hey $playerName$. How's it going?</ts>
        <ts>I've got #numApples# apples since I went and punched that apple tree.</ts>

        <ts>Well, see you later.</ts>
        <actorChangeDirection a="#targetActor#" d="#targetActorDirection#"/>
        <actorMoveTo a="$aId$" x="$x$" y="$y$" z="$z$"/>
    </b>
</Dialog_Script>
```

In the above example, you can see instances of both variables in dialog text as well as in tag attributes. Text tags can contain variables at any point in the text, and any number of them can be included. Tag attributes only allow a single variable in the input at a time, and expects the first and final values of the string to be $ or #.

The dialog system performs error checking for variables requested by the user. For instance, if the tag `jmp` requires an int for its attribute `id`, and it receives a string instead, the dialog execution will abort with an error. Similarly, if the system cannot find a variable with that name, it will also abort.

1.7.4 Tag Types

Script Tags

The dialog system allows the user to execute squirrel scripts from within the execution of a dialog script. Script tags are a non-blocking tag. Below is an example:

```
<Dialog_Script>
<script path = "res://iceCreamScript.nut" id = "0"/>
<b id = "0">
    <ts>I love rum and raisin ice cream!</ts>
    <script id = "0" func = "givePlayerIceCream" v1="$playerId$" v2="20" v3="
↪#iceCreamType#" v4="40"/>
    <script id = "0" func = "printSomething"/>
</b>
</Dialog_Script>
```

Contents of `iceCreamScript.nut`

```
function givePlayerIceCream(playerId, amount, iceCreamType, cost){
    print(playerId + " Got some icecream!");
}

function printSomething(){
    print("This is a function with no arguments!");
}
```

Calling squirrel scripts requires a few things. Firstly, the script must be pre-defined somewhere outside of a block. This is done so that multiple tags can share the same script file.

Secondly, calling a script simply requires referencing the id of the pre-defined script, as well as a function name.

Parameters can be passed to scripts, however these are optional. Up to four parameters are allowed. These parameters are able to make use of the dialog registry system by passing global or local variables to the scripts.

1.8 Input System

The engine contains quite sophisticated support for input devices. As well as the traditional keyboard and mouse, it also contains support for game controllers. This support is implemented by taking an action orientated approach.

1.8.1 Overview

To use the system, there are a few steps that should be followed.

Firstly the actions should be defined:

```
_input.setActionSets({
    "actionSet" : {
        "StickPadGyro" : {
            "Move": "#Action_Move",
            "Camera": "#Action_Camera"
        },
        "AnalogTrigger":{
            "Fire":"#Action_Fire"
        },
        "Buttons" : {
            "Jump": "#Action_Jump",
            "Reload": "#Action_Reload",
        }
    }
});
```

Secondly, they should be mapped to controller or keyboard inputs:

```
//Obtain action handles
::JumpHandle <- _input.getButtonActionHandle("Jump");
::ReloadHandle <- _input.getButtonActionHandle("Reload");
::FireHandle <- _input.getTriggerActionHandle("Fire");
::MoveHandle <- _input.getAxisActionHandle("Move");
::CameraHandle <- _input.getAxisActionHandle("Camera");

//Map keyboard input
_input.mapKeyboardInput(0, ::JumpHandle);
_input.mapKeyboardInput(10, ::FireHandle);
_input.mapKeyboardInputAxis(50, 51, 52, 53, ::MoveHandle); //Designate four keys to␣
↪represent the axis directions.

//Map controller input
_input.mapControllerInput(0, ::JumpHandle); //A Button
_input.mapControllerInput(1, ::ReloadHandle); //B Button
_input.mapControllerInput(0, ::FireHandle); //Left Trigger
_input.mapControllerInput(0, ::MoveHandle); //Left Stick
_input.mapControllerInput(1, ::MoveHandle); //Right Stick
```

Finally, the contents of these actions can be queried.

```
//Returns bool values representing a button press.
local hasJumped = _input.getButtonAction(::JumpHandle);
local shouldReload = _input.getButtonAction(::ReloadHandle);

//Returns a float between 0 and 1 of how far pressed the trigger is.
```

(continues on next page)

(continued from previous page)

```
local fireVelocity = _input.getTriggerAction(::FireHandle);

//Returns a single float each between -1 and 1, with 0 being the middle.
local moveX = _input.getAxisActionX(::MoveHandle);
local moveY = _input.getAxisActionY(::MoveHandle);
```

This action based approach makes mapping inputs easy. The game code checks for "Jump", not for "X button pressed". This means that if the user wishes to use the A button for jumping, they can do so easily.

1.8.2 Mapping Keyboards

While the items within the action sets are named after parts of a controller, the keyboard can still be used to send inputs. This includes buttons, triggers and axises. However it should be noted that the keyboard keys send very rigid values. For instance, when mapping a trigger, the value set will be either 0 or 1. There is no in-between. A similar thing occurs for axises.

1.8.3 Querying by Pressed and Released

The user can filter inputs by providing flags when querying an action.

For instance:

```
//Only returns true when the button is initially pressed.
local pressed = _input.getButtonAction(::AButton, _INPUT_PRESSED);
//Only returns true when the button is released.
local released = _input.getButtonAction(::AButton, _INPUT_RELEASED);
//Returns true when the button is down. The same as providing nothing.
local held = _input.getButtonAction(::AButton, _INPUT_ANY);
```

1.8.4 Querying Specific Devices

Devices are the name given to a source of input, for instance a game controller. The engine supports up to four devices at once. Controllers can be dynamically added and removed during runtime.

Each device is given an id between 0 and 3. The keyboard has its own id. As one device is freed up, it's id can be re-used.

Querying a device looks like this:

```
_input.getButtonAction(::AButton, _INPUT_ANY, 0); //Query device 0
_input.getButtonAction(::AButton, _INPUT_ANY, 1); //Query device 1
//Given how it's checking a different device, one might return true and one might␣
↪return false.
_input.getButtonAction(::AButton, _INPUT_ANY, _KEYBOARD_INPUT_DEVICE); //Query the␣
↪keyboard.

_input.getButtonAction(::AButton); //Not specifying a device will default to the any␣
↪device.
```

1.8.5 The Any Device

The Any Device is the name given to the device which contains data from all devices. Say for instance the user doesn't want to query specific devices for their input. If creating a single player game, they might instead want to listen for any input on a controller or the keyboard. In this case, the any device facilitates this. Any input pressed on any device will be contributed to the any device, with some notes.

Firstly, a button on the any device will return true if any of its contributing devices are true. That is, if the same button is pressed on two controllers, querying the associate action from the any device will return true. Then, if one of the controllers has its button released, it will still return true, as long as a single button is pressed.

At the moment axises and triggers are based on which device most recently set a value. In future that might change.

1.8.6 Setting the Current Action Set

Action sets are set per device. As the action set changes, different actions will receive input based on the current action.

The current action set is set like this:

```
local actionSetFirst = _input.getActionSetHandle("FirstSet");

_input.setActionSetForDevice(0, actionSetFirst); //Set the action set for device 0.
_input.setActionSetForDevice(_KEYBOARD_INPUT_DEVICE, actionSetFirst); //Set for the␣
↪keyboard.
```

Based on which actions were mapped to which buttons of the device, setting the current set will change which actions are sent. An example would be if there was one set for gameplay controls, and another for menu controls. With this method, the user is able to define actions such as 'attack' and 'jump' for the game controls, and menu specific actions such as 'up' and 'down' for the meny system. When the user brings up a menu, the action set for the device would change, and the appropriate actions would be sent.

1.8.7 The Default Action Set

The engine provides a default action set for convenience. It is enabled by default, although can be disabled if the user wishes to define their own. It has the following structure:

```
Default{
    StickPadGyro{
        "LeftMove"
        "RightMove"
    }
    AnalogTrigger{
        "LeftTrigger"
        "RightTrigger"
    }
    Buttons{
        "Accept" //A
        "Decline" //B
        "Menu" //X
        "Options" //Y
        "Start"
        "Select"
        "DirectionUp"
        "DirectionDown"
```

(continues on next page)

(continued from previous page)

```
        "DirectionLeft"
        "DirectionRight"
    }
}
```

If the user does not wish to use this action set, they can create their own by calling the `_input.setActionSets` function. The engine also allows definition of a flag in the `avSetup.cfg` file, if they do not wish to use it. This is for the sake of efficiency, as setting the action set will consume cpu on startup.

```
UseDefaultActionSet false
```

1.9 Sequence Animations

Sequence Animations are a type of animation the user can create to bring interesting effects to their projects. It is intended to be a generic, keyframe based approach to animating. Users can define their animations in an easily edited XML format. The design of the animation system is intended to be flexible and simple, and many different objects can be animated at once.

Sequence animations can be used to create anything from cutscenes to simple position and scale animations for gameplay.

1.9.1 Overview

Below is an example of an animation defined as part of an animation file.

```
<AnimationSequence>
    <data>
        <firstNode type='SceneNode'/>
        <secondNode type='SceneNode'/>
    </data>
    <animations>
        <animMultiPos repeat='false' end='120'>
            <t type='transform' target='0'>
                <k t='0' position='-10, -10, 0'/>
                <k t='120' position='0, 0, 0'/>
            </t>
            <t type='transform' target='1'>
                <k t='0' position='10, 10, 0'/>
                <k t='120' position='0, 0, 0'/>
            </t>
        </animMultiPos>
    </animations>
</AnimationSequence>
```

Firstly, data is defined for the animation. Each piece of data has a target type, in this case a SceneNode. When the user goes to create their animation instance in code, they will supply it with these objects.

Secondly, the animations are created inside the 'animations' tag. Animations are made up of 'tracks'. Tracks effect a single piece of animation data, which is defined in the target attribute.

From there, keyframes are defined within tracks, each with a time value and a piece of data. The pieces of data, for instance 'position' are what are actually animated.

With this multi-track system, the user can animate a number of objects as part of the same animation.

Initiating an animation is done like this:

```
//Provide the path to the animation script.
_animation.loadAnimationFile("res://AnimationScript.xml");

//Create the animation info. It must match what was defined for the animation.
//This same info object can be used to construct many animation instances.
local animationInfo = _animation.createAnimationInfo([firstNode, secondNode]);
//Provide the name of the animation from the file and the info.
//The animation is reference counted and destroyed when the references reach 0.
local currentAnim = _animation.createAnimation("animMultiPos", animationInfo);
```

A component also exists to assign an animation to an entity. With this the animation will persist while the entity exists.

```
_component.animation.add(entity, animationInstance);
```

1.10 Event System

Events are used in the engine to keep track of when something of interest happens. The engine allows scripts to take advantage of this system to call code when they need to. There are two types of event in the engine, system events and user events.

System events are provided by the engine, and are utility things like 'WorldCreated', 'InputDeviceAdded', 'MapChanged', to name a few.

User events are arbuitrary events which can be defined by the user. Any integer id can be given to reference the event, and some arbuitrary data can be provided. The user would want to use user events for things like 'PlayerDied' or 'TankDestroyed'.

1.10.1 Subscribing to System Events

```
function windowResize(id, data){
    print("window resize");
    print(data.width);
    print(data.height);
}

_event.subscribe(_EVENT_SYSTEM_WINDOW_RESIZE, windowResize);
//Unsubscribe that function from that event.
_event.unsubscribe(_EVENT_SYSTEM_WINDOW_RESIZE, windowResize);
```

System events come with a pre-set value. The data provided by the callback function changes depending on the event.

Only one function can be assigned as the receiver for each event type. Attempting to set a second will override the previous.

1.10.2 Subscribing to User Events

User events are subscribed and transmitted like this:

```
enum EVENT{
    PLAYER_WEAPON_CHANGED
};
```

(continues on next page)

(continued from previous page)

```
function weaponChanged(id, value){
    __playerHealth.setText("Weapon changed to: " + value);
}

_event.subscribe(EVENT.PLAYER_WEAPON_CHANGED, weaponChange);
_event.unsubscribe(EVENT.PLAYER_WEAPON_CHANGED, weaponChange);

_event.transmit(EVENT.PLAYER_WEAPON_CHANGED, _currentWeapon);
```

The enum is not necessary, although it is quite useful for code quality. Unlike system events, any number of functions can be assigned to a user event type.

During transmission, any data type can be passed as the second option, including null.

1.11 Scene Querying

The engine allows scenes to be queried using raycasts.

Raycasting is a technique which sends a ray along a specific direction from a specific point, reporting back which objects it hits along the way. This technique can be used for all sorts of functions from gameplay to scene editing.

Note: The user is only able to cast rays in the scene during a designated 'safe' period. This is the 'sceneSafeUpdate' function in the Squirrel Entry File. In this period, the scene cannot be manipulated (nodes moved, scaled. Entities moved etc). This is to ensure that the scene is clean and all transforms have been updated before rays are cast.

1.11.1 Casting a Ray in the Scene

The following snippet is a complete Squirrel Entry File.

```
function start(){
    print("Squirrel Entry file!");
}

function sceneSafeUpdate(){
    if(_input.getMouseButton(0)){
        local posX = _input.getMouseX().tofloat() / _window.getWidth();
        local posY = _input.getMouseY().tofloat() / _window.getHeight();

        local ray = _camera.getCameraToViewportRay(posX, posY);
        local result = _scene.testRayForSlot(ray);
        print(result);
    }
}
```

This is a query of the scene, not of any physics worlds.

This code casts a ray from the current position of the camera in the scene in whatever direction it is facing. This is useful for picking specific objects in the scene with the mouse.

Ray queries can only be called in the sceneSafeUpdate function. This function guarantees that the scene has no dirty flags set, meaning rays can perform correctly. Calling any functions that manipulate the scene tree will throw an error in this function.

1.11.2 Masked Raycasts

Raycast queries can use a masking system to limit the number of responses they return. This can be useful if you need to exclude the number of items to be tested.

The following shows how to apply masks to objects.

```
local firstItem = _scene.createItem("cube");
local secondItem = _scene.createItem("cube");

//Give your objects a bitmask id.
firstItem.setQueryFlags(1 << 6);
secondItem.setQueryFlags(1 << 5);
```

Then, you can test the scene, asking only for items with the specific mask.

```
local foundPos = _scene.testRayForSlot(::createdRay, 1 << 6);
```

The query system is based around bitmask and. This means that if a single bit of the object matches the mask the object will be considered by the raycast.

CHAPTER 2

Creating Content

Creating content for the engine is entirely data driven. The engine contains a number of pre-defined places where content can be setup for use as part of a project. These most commonly include:

- Script files
- Maps defined in the maps directory

All of the content for the engine can be created by hand. However tools exist to help the user create content. Southsea is a tool specifically to design maps/levels for the engine, and should be investigated if you are looking to create or edit maps.

2.1 Maps

Maps are used in the engine to define content for the world. Maps are a fundamental part of the world, and are defined in a data driven way.

Much of the definition of maps are done within the file system, in the *The Maps Directory*.

2.2 Directory Structure

2.2.1 Map Name Directory

The existence of maps is based on the existence of directories within the maps directory. Say for instance the user had the map 'overworld' currently set, the engine would start searching for that maps content within the path:

```
{MapsDirectory}/Overworld
```

The {MapsDirectory} would be replaced with the path to the maps directory. Essentually if there is no directory within the maps directory with that exact name, the engine will load just blank chunks.

2.2.2 Chunks

Chunks themselves are then defined within the map name directory. The name of the directory depends on the coordinates of this chunk, for example `00010001`. That directory would represent the chunk x:1, y:1.

At the moment there are four digits to represent each coordinate, with the x axis being the first four, and the y the second. Similarly to the map name directory, if no directory is found for that chunk a blank chunk will be inserted.

2.3 File Structure

If a chunk is requested for load and the engine finds the correct directory structure for the requested chunk, it will then start loading the files within. These files are what actually define the content of the chunk. These are:

- `meshes.txt` - Static meshes which never move through the lifetime of the chunk, for instance these might be things like rocks, foliage, and so on.
- `bodies.txt` - Static physics shapes which don't move. This can be used to define physics data for things like the meshes in the meshes file.

As well as this, within the chunk directory is the `terrain` directory, which is used to define terrain.

2.4 Terrain Directory

The terrain directory exists within the chunk directory. It is used to define a piece of terrain that should exist within that specific chunk.

Within the terrain directory a few files will exist

- `height.png` - An image used to represent the height of the terrain. This should be a grayscale image. This file is necessary for loading terrain. Without it no terrain will be created.
- `shadow.png` - A pre-baked shadow map.
- `datablock.material.json` - The datablock is the material that's applied to the terrain. It defines things like blend blend layer textures, colours and other things. The datablock is slightly more complex than the other files, so it has a further section below.

Determining whether or not that piece of terrain will be loaded or not is based on a few factors.

- Whether this directory exists in the first place. If the terrain directory isn't within the chunk directory no terrain will be loaded. So if you don't want terrain in your chunks don't create this directory.
- Whether or not the heightmap exists. If it doesn't the terrain won't be loaded.

2.4.1 Terrain Datablock

Datablocks are used by ogre to represent 'materials'. The terrain datablock is a material which is applied to a terrain to define things like blend textures, blend maps and so on. The user doesn't have to define a datablock, as if one isn't found a default will just be used.

Defining a datablock does not depend on a filename at all. The datablock file could be named anything, as long as it ends with the `.material.json` extension. The important part is what's inside the files.

An example of file contents would be:

```
{
    "Terra": {
        "TerraTerrainMap00000000": {
            "diffuse": {
                "value": [
                    1.0,
                    1.0,
                    1.0
                ]
            }
        }
    }
}
```

The file contents is just a json. The most important part of this file is that it is contained within the `Terra` object and has a name referencing the chunk. The Terra object makes sure that this datablock is defined within the terra hlms system, and is necessary for the datablock to be loaded correctly. So in the example here the datablock has the name `TerraTerrainMap00000000`. Ogre relies on datablocks having unique names, and this even includes across hlms systems. To avoid conflicts between the engine, the datablock is prefixed with the `Terra` string.

The engine works by generating a name from the chunk coordinates. It then tries to find a loaded datablock with this name. If for whatever reason this cannot be found it will simply use the default. The breakdown of a datablock name is this:

- `Terra` - Always there. Used as a prefix to avoid conflicts between other hlms'es.
- Map Name - The name of the map. In the example above the map is named 'TerrainMap'.
- Chunk Coordinates - The same as the coordinates for the chunk directory.

If the datablock is named anything else it won't be applied to the terrain.

Note: Internally the engine creates a new ogre resource group in the terrain directory when the load begins. This is used to avoid conflicts between images such as `height.png`.

However, datablocks don't follow this convention. Instead if two datablocks have the same name an assertion is thrown. Furthermore, datablocks aren't unloaded when the resource group is unloaded. This makes it easy to run into assertion errors due to conflicting datablock names if chunks are often being loaded and unloaded, and the names were configured incorrectly.

The engine is setup to try and destroy the datablock name specified above. So if the names have been setup correctly, the destruction of the datablocks will happen correctly.

Warning: Do not define multiple datablocks in the terrain directory! Any datablocks in the terrain directory will be loaded when the resource group is created, but not destroyed when the group is unloaded. It is possible to iterate through datablocks in a certain group and destroy them, however this is a costly operation. The engine does not do this for the sake of efficiency, and instead assumes that the user defined their names correctly.

If you create any other sort of datablock (pbs or terra) you will run into assertions if the terrain directory is loaded twice.

So don't do that.

2.5 Ogre Material Cookbook

Ogre supports pbs and unlit implementations by default.

2.5.1 PBS

Transparency

Simple transparent material.

```
{
    "blendblocks" :
    {
        "waterBlend" :
        {
            "dst_blend_factor" : "one"
        }
    },

    "pbs" :
    {
        "waterMaterial" :
        {
            "workflow" : "metallic",
            "diffuse" :
            {
                "value": [0.1, 0.1, 1]
            },
            "transparency" :
            {
                "value" : 1,
                "mode" : "transparent"
            }
        }
    }
}
```

The transparency of the object is applied based on the "transparency" entry. A value of 1 makes the object more transparent and 0 makes it less.

The blendblock is necessary to make the transparency active.

Alpha Transparency

Rendering a mesh, discarding certain pixels if they contain alpha. This can act as an optimisation, as the scene does not treat the object as transparent, meaning the most efficient sorting can be used. However this can lead to some incorrect depth calculations.

```
{
    "blendblocks" :
    {
        "spiderNestBlend" :
        {
            "src_blend_factor" : "one",
```

(continues on next page)

(continued from previous page)

```
            "dst_blend_factor" : "one_minus_src_alpha"
        }
    },

    "pbs" :
    {
        "SpiderObjectsWebMaterial" :
        {
            "workflow" : "metallic",
            "blendblock": "spiderNestBlend",
            "diffuse" :
            {
                "value" : [1, 1, 1],
                "texture": "SpiderWebParts.png"
            }
        }
    }
}
```

Detail layers

Detail layers are used to provide additional detail to an existing material. For instance, they are useful for showing things like dirt or blemishes on a character's face, where the standard diffuse and normal textures have been used for the face itself. Detail layers allow themselves to be tiled, scaled and offsetted, which makes them useful for animation.

This material uses two normal maps to produce a scrolling water effect.

```
{
    "blendblocks" :
    {
        "waterBlend" :
        {
            "dst_blend_factor" : "one"
        }
    },

    "pbs" :
```

(continues on next page)

(continued from previous page)

```
    {
        "waterMaterial" :
        {
            "blendblock": "waterBlend",
            "workflow" : "metallic",
            "diffuse" :
            {
                "value": [0.1, 0.1, 1]
            },
            "detail_normal0":{
                "value" : 2,
                "texture": "SeaPattern.tga"
            },
            "detail_normal1":{
                "value" : 1,
                "texture": "SmallWaves.tga"
            },
            "transparency" :
            {
                "value" : 1.0,
                "mode" : "Transparent"
            }
        }
    }
}
```

Tiling Textures

Sampler blocks can be used to tile textures. By default they will be clamped. Detail diffuse is used because it allows scale to be specified.

```
{
    "samplers" :
    {
        "wrapSampler" :
        {
            "u" : "wrap",
            "v" : "wrap",
            "w" : "wrap"
        }
    },

    "pbs" :
    {
        "testingFloor" :
        {
            "workflow" : "metallic",
            "detail_diffuse0" :
            {
                "scale": [15, 15],
                "texture": "checkerPattern.png",
                "sampler": "wrapSampler"
            }
        }
    }
```

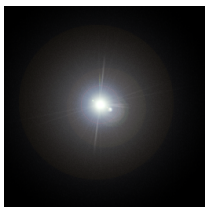
(continues on next page)

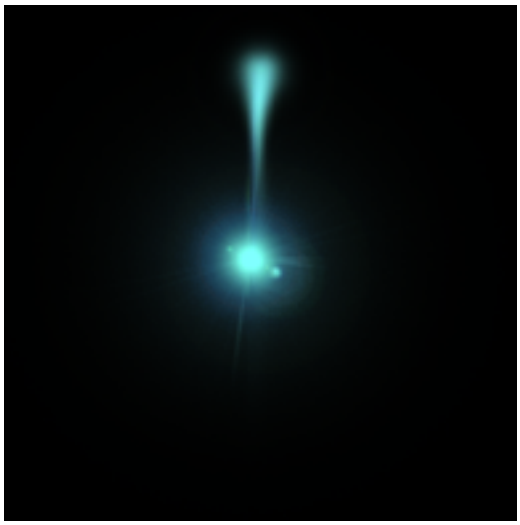
(continued from previous page)

```
}
```

2.5.2 Unlit

Diffuse Layers





Unlit datablocks can be decorated with different layers.

```
{
    "unlit" :
    {
        "colourExample" :
        {
            "diffuse": [0.5, 1, 1, 1],
            "diffuse_map0" :
            {
                "texture": "flare.png",
                "blendmode": "Add"
            },
            "diffuse_map1" :
            {
```

(continues on next page)

(continued from previous page)

```
                "texture": "flaretrail.png",
                "blendmode": "Add"
            }
        }
    }
}
```

In this example two images are layered ontop of each other. The blendmode 'Add' is used, which simply adds the current layer's pixel values to the image so far. This means black backgrounds are removed and used as alpha.

Up to 15 diffuse layers can be supplied, meaning lots of combinations can occur.

A diffuse colour is also applied. This is applied after the diffuse maps are processed, in this case giving the final image a blue tint.

2.6 JSON Materials

Ogre allows the user to define materials in a json format. This format is human readable and easily understood, and can be used without needing to recompile the engine.

Ogre contains two hlms implementations out of the box, PBS and Unlit.

2.6.1 PBS

Every option for a PBS json material is shown below:

```
{
    "pbs" :
    {
        "material_name" :
        {
            "macroblock" : "unique_name" ["unique_name", "unique_name_for_
↪shadows"],
            "blendblock" : "unique_name" ["unique_name", "unique_name_for_
↪shadows"],
            "alpha_test" : ["less" "less_equal" "equal" "not_equal" "greater_
↪equal" "greater" "never" "always" "disabled", 0.5],
            "shadow_const_bias" : 0.01,

            "workflow" : "specular_ogre" "specular_fresnel" "metallic",

            "transparency" :
            {
                "value" : 1.0,
                "mode" : "None" "Transparent" "Fade",
                "use_alpha_from_textures" : true
            },

            "diffuse" :
            {
                "value" : [1, 1, 1],
                "texture" : "texture.png",
                "sampler" : "unique_name",
```

(continues on next page)

(continued from previous page)

```
                "uv" : 0,
                "grayscale": true
            },

            "specular" :
            {
                "value" : [1, 1, 1],
                "texture" : "texture.png",
                "sampler" : "unique_name",
                "uv" : 0
            },

            "roughness" :
            {
                "value" : 1,
                "texture" : "texture.png",
                "sampler" : "unique_name",
                "uv" : 0
            },

            "fresnel" :
            {
                "mode" : "coeff" "ior" "coloured" "coloured_ior",
                "value" : [1, 1, 1],
                "texture" : "texture.png",
                "sampler" : "unique_name",
                "uv" : 0
            },

            "metallness" :
            {
                "value" : 1,
                "texture" : "texture.png",
                "sampler" : "unique_name",
                "uv" : 0
            },

            "normal" :
            {
                "value" : 1,
                "texture" : "texture.png",
                "sampler" : "unique_name",
                "uv" : 0
            },

            "detail_weight" :
            {
                "texture" : "texture.png",
                "sampler" : "unique_name",
                "uv" : 0
            },

            "detail_diffuse0" :
            {
                "mode" : "NormalNonPremul" "NormalPremul" "Add" "Subtract"
↪"Multiply" "Multiply2x" "Screen" "Overlay" "Lighten" "Darken" "GrainExtract
↪" "GrainMerge" "Difference",
```

(continues on next page)

(continued from previous page)

```
                "offset" : [0, 0],
                "scale" : [1, 1],
                "value" : 1,
                "texture" : "texture.png",
                "sampler" : "unique_name",
                "uv" : 0
            },

            "detail_normal0" :
            {
                "offset" : [0, 0],
                "scale" : [1, 1],
                "value" : 1,
                "texture" : "texture.png",
                "sampler" : "unique_name",
                "uv" : 0
            },

            "reflection" :
            {
                "texture" : "cubemap.png",
                "sampler" : "unique_name"
            },

            "emissive" :
            {
                "value" : [1, 1, 1],
                "texture" : "texture.png",
                "sampler" : "unique_name",
                "uv" : 0,
                "lightmap": true
            }
        }
    }
}
```

detail_diffuse and detail_normal can have up to four separate entries, where the index is affixed to the end of the declaration.

2.6.2 Unlit

Every option for an Unlit json material is shown below:

```
{
    "unlit" :
    {
        "material_name" :
        {
            "macroblock" : "unique_name" ["unique_name", "unique_name_for_
↪shadows"],
            "blendblock" : "unique_name" ["unique_name", "unique_name_for_
↪shadows"],
            "alpha_test" : ["disabled", 0.5],
            "shadow_const_bias" : 0.01,
```

(continues on next page)

(continued from previous page)

```
            "diffuse": [1, 1, 1, 1],
            "diffuse_map0":
            {
                "texture": "texture.png",
                "sampler" : "unique_name",
                "blendmode": "NormalNonPremul" "NormalPremul" "Add" "Subtract
↪" "Multiply" "Multiply2x" "Screen" "Overlay" "Lighten" "Darken"
↪"GrainExtract" "GrainMerge" "Difference",
                "uv": 0,
                "animate": [
                    [1, 0, 0, 0],
                    [0, 1, 0, 0],
                    [0, 0, 1, 0],
                    [0, 0, 0, 1]
                ]
            },
            "diffuse_map1": {},
            "diffuse_map2": {},
            "diffuse_map3": {},
            "diffuse_map4": {},
            "diffuse_map5": {},
            "diffuse_map6": {},
            "diffuse_map7": {},
            "diffuse_map8": {},
            "diffuse_map9": {},
            "diffuse_map10": {},
            "diffuse_map11": {},
            "diffuse_map12": {},
            "diffuse_map13": {},
            "diffuse_map14": {},
            "diffuse_map15": {}
        }
    }
}
```

Settings for subsequent diffuse maps are omitted here for clarity's sake. The same settings apply to every diffuse map.

2.6.3 Blendblocks, Samplerblocks, Macroblocks

Blendblocks, samplerblocks and macroblocks are used in combination with regular materials to provide extra functionality. Their settings are separated out from the actual material definition to avoid redundancy and to better improve performance when performing similar tasks. Once defined, many datablocks can reference these blocks.

```
{
    "samplers" :
    {
        "unique_name" :
        {
            "min" : "point" "linear" "anisotropic",
            "mag" : "point" "linear" "anisotropic",
            "mip" : "none" "point" "linear" "anisotropic",
            "u" : "wrap" "mirror" "clamp" "border",
            "v" : "wrap" "mirror" "clamp" "border",
```

(continues on next page)

(continued from previous page)

```
            "w" : "wrap" "mirror" "clamp" "border",
            "miplodbias" : 0,
            "max_anisotropic" : 1,
            "compare_function" : "less" "less_equal" "equal" "not_equal"
↪"greater_equal" "greater" "never" "always" "disabled",
            "border" : [1, 1, 1, 1],
            "min_lod" : -3.40282347E+38,
            "max_lod" : 3.40282347E+38
        }
    },

    "macroblocks" :
    {
        "unique_name" :
        {
            "scissor_test" : false,
            "depth_check" : true,
            "depth_write" : true,
            "depth_function" : "less" "less_equal" "equal" "not_equal"
↪"greater_equal" "greater" "never" "always",
            "depth_bias_constant" : 0,
            "depth_bias_slope_scale" : 0,
            "cull_mode" : "none" "clockwise" "anticlockwise",
            "polygon_mode" : "points" "wireframe" "solid"
        }
    },

    "blendblocks" :
    {
        "unique_name" :
        {
            "alpha_to_coverage" : false,
            "blendmask" : "rgba",
            "separate_blend" : true,
            "src_blend_factor" : "one" "zero" "dst_colour" "src_colour" "one_
↪minus_dst_colour" "one_minus_src_colour" "dst_alpha" "src_alpha" "one_
↪minus_dst_alpha" "one_minus_src_alpha",
            "dst_blend_factor" : "one" "zero" "dst_colour" "src_colour" "one_
↪minus_dst_colour" "one_minus_src_colour" "dst_alpha" "src_alpha" "one_
↪minus_dst_alpha" "one_minus_src_alpha",
            "src_alpha_blend_factor" : "one" "zero" "dst_colour" "src_colour
↪" "one_minus_dst_colour" "one_minus_src_colour" "dst_alpha" "src_alpha"
↪"one_minus_dst_alpha" "one_minus_src_alpha",
            "dst_alpha_blend_factor" : "one" "zero" "dst_colour" "src_colour
↪" "one_minus_dst_colour" "one_minus_src_colour" "dst_alpha" "src_alpha"
↪"one_minus_dst_alpha" "one_minus_src_alpha",
            "blend_operation" : "add" "subtract" "reverse_subtract" "min"
↪"max",
            "blend_operation_alpha" : "add" "subtract" "reverse_subtract"
↪"min" "max"
        }
    }
}
```

2.7 Southsea

Southsea is a level editor created specifically for the avEngine. It provides a range of functionality specifically to create maps for the engine. Its main features include:

- A feature full terrain editor, including height and blend painting.
- A convenient workflow to edit the engine's chunk based worlds.
- Easy integration with pre-existing avEngine projects (Able to edit a project from any avSetup.cfg file).
- Complete support for editing objects within the scene tree.
- General creature-comfort utilities such as undo and redo, common key-press functionality, drag and drop, so on.
- Customisable user interface with dockable windows.
- The ability to view a map in the engine with a button press.

Southsea is the encouraged method for editing project maps in the avEngine. Its interface has been designed to follow a number of conventions set by pre-existing level editor tools.

It takes care of much of the setup of maps and map files. It generates the appropriate files and textures based on user edits.

2.8 Generating Meshes

Producing meshes for use in the engine is an important consideration.

The avEngine makes use of Ogre3D, and as such all meshes are provided in Ogre's mesh format. Due to this, all of Ogre's tools and documentation can be used to produce suitable meshes.

This document will explain how to setup a suitable environment for mesh creation, and explain how to export them to the engine.

2.8.1 Blender

Blender is the recommended application for 3d modelling. The ogre project provides a plugin named blender2ogre, which can be downloaded from their repositories. When installed this will enable an export option in blender to export to ogre.

In order to install this project in Linux, copy the directory `io_ogre` from their repository into the blender addons folder. Consult Blender's documentation to find this location, but in linux this path is generally:

```
~/.config/blender/2.83/scripts/addons
```

`2.83` changes depending on what version of blender you have. If this path doesn't exist for you you can create it.

It is recommended to create the `io_ogre` directory as a symlink, so that if you ever pull some changes from the blender2ogre repository they will be updated automatically.

```
ln -s ~/blender2ogre/io_ogre ~/.config/blender/2.83/scripts/addons/io_ogre
```

The blender2ogre addon requires use of ogre's mesh tool. This is a binary tool included with your ogre distribution that allows exporting of xml files to ogre's mesh format.

The purpose of the xml format is to provide an intermediate file type, where vertices, faces and materials are described in plain text. These files have a `.mesh.xml` extension.

Ogre can't read these formats directory however. It is only able to read a compressed binary `.mesh` file. The blender2ogre extension produces an xml file, and then uses the ogreMeshTool to produce the final .mesh format.

In order to use the extension, edit the file `io_ogre/config.py`. Find the line containing the string `OGRETOOLS_XML_CONVERTER` and replace it with the path to your ogreMeshTool. The avEngine uses ogre 2.x, so the binary file will be called ogreMeshTool, not OgreXmlConverter.

The tool will be included in your built ogre distribution, under the path `ogre/build/bin/OgreMeshTool` Before using it in blender it is worth checking that you can run the converter without any issues. If you have ogre installed on your path, you shouldn't need to change this config value.

Back in blender, check the new extension is loaded, by opening edit>preferences>addons, and search for ogre. Enable this extension, and test it out by exporting a mesh. If you have problems with the mesh exporter blender should alert you of this.

If you still have problems with the meshTool path, blender does allow you to set it using their gui, inside the extensions window. Expand the options for the mesh tool, and you should see it there.

2.8.2 Every created skeleton animation is named 'my_animation'

Name your animation whatever you want in the hierarchy view on the left.

Switch to the dope sheet view, then the action editor. Ensure your current track is selected in the options. There should be an option somewhere on the left with a snowflake with the word 'stash' next to it.

Pressing this will register your animation in the NLA stack.

2.8.3 avTools asset pipeline

The asset pipeline created for the engine includes support for automatic exporting of blender files to .mesh format.

Before using it you should setup your default settings in the blender ogre exporter window.

- Set swap axis to x-zy (ogre default)
- Disable export materials
- Disable export scene

2.9 Asset Pipeline

One prime consideration for content creation is managing assets. If no structured approach is created, managing the creation and lifetime of assets can quickly become uncontrollable.

A number of considerations exist:

- Will there be a separation between project files and raw assets (i.e blender project to .mesh)
- How will assets be stored and versioned.
- What if something small needs to be changed?

One of the most common sources of slowdown is exporting assets to their correct location. With the example of meshes, if done manually, the user would have to open the 3d modeller project file, click file>export, choose a directory, click export. Doing this manually for even a small change is error prone and time consuming. This concept could be extended to other types of resources such as textures. For instance, the user might be creating assets in a vector format, but expected to rasterise that for the final project. In this instance, the vector files should be used as the original source of the asset, not the rasterised image.

Regarding the question of what should be versioned, it is incorrect to version the final .mesh file. Really this file is an output of the modeller application. Instead, it should be the modelling application's project file that is versioned, and then the final .mesh file is produced from this. Separating project files from their output helps reduce clutter.

The asset pipeline tool helps fulfil these desires. It is a script which allows the user to automate the output of design tools such as 3d modellers to a format the engine can understand. The entire process should be based around a single command line command. The .blend and .svg files are versioned appropriately, and then the user can run this command to produce a full directory structure containing the assets which can be understood by the engine. There is no need to open blender and manually export assets, as the tool does this by itself.

2.10 User Entity Components

To support the data-driven approach of the engine, the user is able to define their own custom components which can be assigned to entities. These components can contain a number of user definable data types which can be get and set using the component api.

The user is able to define their components in the setup file like this:

```
"Components": {
    "health": ["int"],
    "statusAfflictions": ["int", "float", "bool", "float"],
    "coinAmount": ["int", "int", "int", "int"]
}
```

When the components are defined, the user can access them like this:

```
enum Components{
    HEALTH,
    AFFLICTIONS,
    COIN
};
enum HealthVariables{
    VALUE,
};

local en = _entity.create(SlotPosition());
_component.user[Components.HEALTH].add(en);

_component.user[Components.HEALTH].set(en, HealthVariables.VALUE, 50);
local health = _component.user[Components.HEALTH].get(en, HealthVariables.
↪VALUE);
```

2.10.1 Variable types

The user has three data types to choose from, `int`, `float` and `bool`. The user is allowed 4 variables per component. This limit is put in place for performance reasons.

Similarly, a maximum of 16 components can be defined. Components are defined in the setup and cannot be changed during the execution of the engine.

2.10.2 Accessing components

Components are id'd using a numeric identifier.

The user does not have to use enums, for example:

```
_component.user[0].set(en, 0, 50);
```

The above code is perfectly valid. Components are id'd based on their index in the list when they were defined, and the same goes for their variables. None of them are defined by string value, again for performance reasons, so the user is encouraged to use enums to make their code more descriptive.

CHAPTER 3

Setup

3.1 Engine Startup

When the engine comes to start itself up there are a number of steps it goes through. Much of this can be configures to suit the needs of the user, and the engine itself takes a data driven approach when deciding how the setup should happen.

Things the engine needs to take into account would include:

- Where are the data files for the game?
- Where are the files necessary for libraries to function (HLMS shaders for instance)?
- Where should things like save files be stored?
- Other useful settings like window titles.

Bear in mind that many of these files and requirements fall into different categories. For instance, Ogre's HLMS shaders are necessary for ogre, but have nothing to do with the game files themselves. Therefore they shouldn't be stored with the game files. Furthermore, save files are written by the engine, but have nothing to do with the game files either, so they should target a separate directory. The engine makes lots of distinctions like this and allows the user to specify their exact settings.

The steps taken in the startup to determine settings are outlined below:

- Start the `Base` class.
- Determine the Master Directory.
- Check for an `avSetup.cfg` file.
- Read in the settings from the avSetup file.

3.1.1 Directories

The engine has a number of specified directories (folders) which it expects to find in order to execute correctly. Some are more important than others.

Av directories:

- Master Directory
- Data Directory
- Save Directory

Master Directory

The Master Directory is the most important of the lot, as without it the engine will immediately terminate. It is the directory in which the engine expects to find its engine files and config files. It is the directory which contains the avSetup file.

The master directory is the only hard coded directory which can't be changed by config or settings. On Linux and Windows the master directory is the present working directory (pwd), which is just the directory in which the engine is ran from. On MacOS, the Master Directory is set to be the Resources directory inside the app bundle.

Data Directory

The Data directory is the directory which actually contains the game files. I like to refer to these as the data files, which is why it's called the data directory. The Data Directory is entirely built up of game files, which should be read only.

For more information please see *Data Directory*.

Save Directory

The Save Directory is the directory in which the serialiser will write its contents to. This should be the only directory which the engine writes to.

3.1.2 Engine and Game files

There is a big difference between the purpose and categories of files. Some are engine specific files, while some others are game specific.

Engine files are used for the execution of the engine, and are universal to all games powered by the engine. An example would be the HLMS shader files for Ogre.

Game files are files that the engine loads in to represent the game.

The difference between the file types means the same binary engine can power a number of different games without the need to rely on ridgidly defined paths in the engine code.

3.1.3 avSetup File

The `avSetup.cfg` file is a game file. There is intended to be one per project built with the engine.

The file is an important file in the setup of the engine. It is responsible for outlining many of the basic options used by the engine, such as paths to other resource directories, and info such as a window title.

More details of this file are provided in the section <section>

3.1.4 Ogre HLMS files

The Ogre HLMS files are a group of files necessary for the correct operation of Ogre. They're used as building blocks to generate shaders. They are engine files, and are therefore defined as part of the code. They are expected to be in the master directory under the name of `Hlms`, and are copied into the master directory by the build system from the ogre directory. If the engine can't find them it will abort.

3.2 Data Directory

The data directory is one of the fundamental places in which data should be stored. Files in the data directory are read only, and are used as actual resources for the execution of the game.

The data directory is a loose term for a collection of files. The engine has some fundamental contents which the engine looks for, and are built into the code.

For instance:

- The squirrel entry file. Default `squirrelEntry.nut`
- The Ogre resources file. Default `OgreResources.cfg`
- The maps directory. Default `maps`

The exact location of these files and directories can also be specified directly in the `avSetup.cfg` file, however it is much more convenient for the user to just specify the path to the data directory. The engine will then search for these files based on the above defaults. If any of these files aren't found the engine won't crash, however it will start in a state where certain things might not work. For instance, if no squirrel entry file is provided the engine will launch but do nothing.

Note: The data directory is automatically assumed to exist in the master directory, unless otherwise stated in the `avSetup.cfg` file.

If you want the data directory to be inside the same directory as the avSetup.cfg file, you must specify that in the file. For example:

```
DataDirectory    .
```

All paths within the setup file are resolved relative to the setup file's path.

3.2.1 Squirrel Entry File

The squirrel entry file is the first squirrel script that gets executed by the engine. The engine itself isn't hard coded to run any other script files unless told to. The purpose of the entry file is to tell the engine how it should start up, and acts like the 'main' function in other programming languages.

In an actual game it would be used to do things like setup the world, load in save state and kick off the actual game execution. However, in something like a test it can just be used to setup state and assert a pass or failure value. Conversely, if the user just wanted to see the engine do something, the user could just make it load a model and move the camera to be able to see it.

3.2.2 Ogre Resources File

The ogre resources file is a file that registers the ogre resource locations. Ogre itself has its own resource manager that deals with things like textures and meshes. The user can supply these resource locations with this file. It is a simple config file which contains various paths.

```
[General]
FileSystem=/home/edward/Documents/avData/meshes
FileSystem=../resourcesDirectory
```

In this example I provide an absolute path and a relative path to directories. The engine accepts both. Relative paths are resolved relative to the location of the ogre resources file, **not** the data directory. This is more manageable for tests.

3.2.3 The Maps Directory

The maps directory is a directory which contains maps. These maps are contained inside of a directory structure which in itself describes them.

3.3 AvSetup File

The avSetup file is a crucial metadata file used by the engine. It contains metadata which describes details of the project, and outlines how the engine should start itself.

The intention of the avSetup file is to allow customisation of how the engine loads itself. Given its data driven approach, the engine would be capable of powering a number of different games without having to change the binary at all. The intention of the avSetup file is to specify where exactly it should look to find this data.

Say for instance you were working on a modified version of a set of game files, and you didn't want anything you did to interfere with them. You could place them somewhere separate from the engine install, all you would have to do to load these other files in, rather than the supplied ones would be to provide a different config file.

The config file is intended to represent a single game to be powered by the engine. Multiple config files can be used to represent multiple games, and allow easy swapping between games, rather than having to re-compile things. This also alleviates the trouble of having to hard code search paths into the engine binary, as these things can be fed in at runtime.

The avSetup file is also used by the testing framework to describe a single test. It provides an easy way to encapsulate a single test case without having to modify the engine at all, as during startup the engine can just be pointed to a different file which represents a different test case.

3.3.1 Contents

The file itself contains JSON data. An example of a setup file is shown below

```
{
    "WindowTitle": "Empty",
    "CompositorBackground": "1 0 1 1",
    "SquirrelEntryFile": "squirrelEntry.nut",
    "DataDirectory": ".",
    "MapsDirectory": "../common/maps"
}
```

In this example, the file specifies aspects of a simple project. Each value is later parsed by the engine and used as part of setup.

All path values specified are relative to the setup file.

Basic setup file values

Value	Meaning
WindowTitle	The title which will be set for the render window.
DataDirectory	A path to the data directory. This path will be resolved relative to the setup file.
CompositorBackground	A colour value which is used for the default background colour. Should be in the format '1 0 1 1', where each value represents one of RGBA.
OgreResourcesFile	A path to a file containing ogre resource locations.
SquirrelEntryFile	A path to the script file which will be executed at the start of the engine execution.
MapsDirectory	A path to the directory which contains map data.
SaveDirectory	A path to a directory which contains saveable data. This should be writable.
WorldSlotSize	An integer value which outlines how big a slot in the world is.
WindowResizable	A boolean specifying whether the user should be able to resize the window.
WindowWidth	The initial width of the window.
WindowHeight	The initial height of the window.
DialogScript	A path to the script file which will be called for dialog events.
UseDefaultActionSet	Boolean of whether the engine should setup the default action set.
UserSettings	A json object containing a list of user settings.

Setup file values for testing

These values are related specifically to testing. If the engine was built without testing capabilities, these settings are ignored.

Value	Meaning
TestMode	True or false depending on whether this is a test.
TestName	The name of the test.
TestTimeout	An integer representing the number of seconds until the test should timeout. This is used to prevent a broken test never failing.
TestTimeoutMeansFailure	True or false for how the engine should treat a timeout. Projects such as stress tests might not want to fail on test timeout.

3.3.2 User Settings

The user is able to set custom settings values for their project. These values can later be read as required.

An example of this syntax is shown below:

```
"UserSettings": {
    "userInt": 100,
    "userFloat": 100.1,
    "userBool": true,
    "userBoolCase": false,
    "userString": "Some example text",
    "SecondSectionInt": 300
}
```

These values can be read from scripts like this:

```
local userValue = _settings.getUserSetting("userValue");
```

The returned type depends on the value set in the file.

CHAPTER 4

Squirrel

4.1 Squirrel Overview

Squirrel is a scripting language library used by the av engine. It is leveraged to provide a flexible and powerful facility for data driven interaction with the engine. A number of functionalities of the engine are exposed through the scripting language. It allows the implementation of gameplay aspects without the need to touch any of the c++ code or recompile the engine.

The engine is entirely built around squirrel as a means for implementing functionality and heavily encourages its use.

4.1.1 APIs and Functionality

The engine exposes apis for squirrel to call. These act as handles to c++ functions, which allows c++ functions to be called by scripts.

The api has been designed in a way which is easy to use but is still powerful in its capabilities. The apis are exposed to squirrel using a namespace approach. Api calls will look something like this:

```
_mesh.createMesh("ogrehead2.mesh");
_camera.setPosition(0, 0, 0);
local e = _entity.create();
```

As you can see, the api wraps individual calls behind namespaces such as `_mesh` or `_camera`. The intention of this is to group the function calls together into their own individual namespaces. This would prevent me from having to otherwise write something like `_meshCreateMesh`, which looks ugly.

Tip: Each namespace is prefixed by an _. The purpose of this is to make it clear that this is indeed a function call rather than anything else.

It also helps to prevent name collisions. Consider this example:

```
local mesh = _mesh.createMesh("");
```

In that example a mesh is created with the name mesh. That's quite a common name, but now consider that I didn't include the _.

```
local mesh = mesh.createMesh("");
```

We now have a collision, as mesh has been re-assigned to mean a reference to a mesh. The entire namespace is now broken. The _ character helps make sure this sort of situation never arises.

Warning: You might be thinking that calling:

```
local _mesh = _mesh.createMesh("");
```

Would do the same thing, and you'd be right! At the moment there is no protection against overriding the namespace, so please try and avoid it. Not to mention that if one script does it the whole thing will break for other scripts as well. So don't do that.

All of the apis documented in the further sections follow a similar approach. You can refer to the title of the page to find out which namespace those functions fall under.

4.1.2 Engine Specific Types

SlotPosition

The engine exposes a few common built in types to squirrel. These can be used within squirrel as classes, and constructed in a way that is easy to understand. For example:

```
local position = SlotPosition(1, 2, 10, 20, 30);
```

The above line of code will create an instance of a SlotPosition. This SlotPosition behaves identically to the SlotPosition found in the c++. It follows the same rules of overflow and underflow, as well as the origin respecting conversion.

```
local first = SlotPosition(1, 2); //Just the slot coordinates.
local second = SlotPosition(3, 4, 50, 60, 70); //Slot coordinates and a position.

local third = first + second;
third.toVector3(); //Translate relative to the origin.
```

Vector3

Most of the time the engine does not provide a means to represent a vector. This is purley for the purpose of efficiency. Wrapping functionality around a class can become quickly convoluted, and for something simple like a vector, an array for representation is much more efficient. The SlotPosition requires its own class and container because in reality a slot position is a much more complex data object than a vector. Sanity checks and shifting is necessary for SlotPositions, while not necessary for vectors.

Furthermore, the engine often times will take plain values for function parameters rather than something like a vector3 object. This is again for the sake of efficiency, as providing three floats to represent a vector3 is much more efficient than providing a wrapper class.

```
local result = SlotPosition(1, 2, 10, 20, 30); //Here there is no separator between␣
↪the slot positions and the actual positions.
local vec = first.toVector3(); //Returns an array.
```

(continues on next page)

(continued from previous page)

```
print("x: " + vec[0]);
print("y: " + vec[1]);
print("z: " + vec[2]);
```

4.1.3 Squirrel Entry File

The first script executed is the squrrel entry file. This script is responsible for the startup of the engine, and is therefore very important.

The entry file is provided based on information in the `avSetup.cfg` file. For more information please see *Squirrel Entry File*.

4.1.4 Scripted States

Scripted states are a way in which the user can specify scripts to run. Say for instance the user was in a situation where they had a script they wanted run each frame for a little while. For instance, if the weather in the game became rainy. If in this situation some sort of water based entity should start spawning, how would that be implemented?

Scripted states solve this problem. With a scripted state you can specify a script to be run each frame. The user has complete control over when the state starts and when it ends. The script will be run until the state is ended. In the case of the rain example, the state would be started when the rain starts, the script would create monsters as it runs, and then stop when it ends.

Scripts work by providing a name for the state and a script to run. This script has three functions which are run, `start()`, `update()` and `end()`. These functions are expected to be defined within the provided script file. Start is called once when the state begins, end is called once when it ends. Update is called each update tick until the state is ended.

So in the above described example, update would generate random numbers to decide when to generate monsters, and end would kill them all off horribly, because they can't survive without the rain. If code for start is not needed, this function can be omitted. The same applies to the other functions.

There are a number of places where states become useful, for instance boss fights might want a state to run until the boss ends. This state might print things to the screen, or generate hazards for the player.

If the player was in one of those levels where if you walk into an area you get found out, a state could be made to keep track of whether the player has gone there yet.

The flexibility of being able to specify a script for the task makes the system very powerful. As a matter of fact, the squirrel entry file is being run as a state, meaning you can define the start, update and end functions in your script and have them run. This is a state known as the `EngineState`, and is the only state which cannot be ended manually. It will be started on engine start, and ended on engine shutdown. If there are any states left running during engine shutdown, they will be ended.

4.1.5 Script File Types

The c++ has two different ways to compile and execute a squirrel file. These are the `Script` class and the `CallbackScript` class. These two classes are used to compile squirrel files and expose them to the c++, but they do this in different ways. The Script class compiles scripts as a single executable. The Callback script compiles them into multiple closures, which can be individually executed on request. The Callback script class is used to perform the various callback operations that the engine performs.

4.2 Squirrel API

Here is the documentation for the squirrel api calls.

4.2.1 Function Namespaces

_camera

The camera api gives access to functions regarding the camera. There is a single camera in the engine, which makes it easier to call these functions, as no camera reference needs to be passed as an argument.

Example

```
_camera.setPosition(0, 0, 100);
_camera.lookAt(0, 0, 0);
```

API

setPosition (*x*, *y*, *z*)
Set the position of the camera.

lookAt (*x*, *y*, *z*)
Set the camera to look at a specified position.

_mesh

The mesh interface is used to create simple meshes and insert them into the ogre scene. They are not intended for gameplay usage, but rather for simple debugging. If the user wishes to see 'something on the screen' this will be a useful interface for them.

Meshes do not interract with the world in any way, and each mesh sits on a separate scene node from the root.

Example

```
local s = _mesh.createMesh("ogrehead2.mesh");
_mesh.setPosition(s, 0, 30, 0);
_mesh.destroyMesh(s);
```

API

createMesh (*meshName*)

Arguments

- **meshName** (*String*) – The name of the ogre mesh resource to be created.

Returns A User Data object representing the mesh.

Create a new mesh in the ogre scene and add it to the origin.

destroyMesh (*mesh*)

Arguments

- **mesh** (*UserData*) – The mesh to destroy.

Destroy a mesh in the scene.

setPosition (*mesh*, *x*, *y*, *z*)

Arguments

- **mesh** – The mesh to set the position of.

Set the position of a mesh in the scene.

_world

Interface for creating and destroying the world.

The engine provides flexibility as to when the user wishes to create a world, and by default one is not created until the `createWorld()` function is called from a script. The world itself is stored as a singleton, meaning there can only be one instance of it at a time. If you wish to create a new world, the old one must be destroyed first.

Example

```
_world.createWorld();
_world.destroyWorld();
```

API

createWorld()

Returns True if the world was created successfully, false if not.

Creates a new world. This function will fail and no world will be created if one already exists.

destroyWorld()

Returns True if the world was destroyed successfully, false if not.

Destroy the world. This function will fail if no world exists.

_entity

The Entity namespace provides access to the Entity Manager. Operating on and effecting entities largely happens in the component namespace or through the EntityClass.

Example

```
local en = _entity.create(SlotPosition());
en.setPosition(SlotPosition(0, 0, 100, 100, 100));
_component.mesh.add(en, "ogrehead2.mesh");
```

API

create (*SlotPosition*)
Create an entity at the specified position.

_input

The input namespace provides access to the simple input system used to detect key presses.

Example

```
if(_input.getKey(_KeyUp)) print("Wow key is up");
```

API

getKey (*keyId*)
Get the boolean value of the specified key.

4.2.2 Class Namespaces

SlotPosition

The SlotPosition namespace is an extension of the c++ SlotPosition. Many of the functions in the api return or take a SlotPosition as an argument.

SlotPositions themselves are relatively complicated, and therefore require their own wrapper class. The SlotPositions in Squirrel follow the same rules and structure of the c++ version.

Example

```
local first = SlotPosition(1, 2);
local second = SlotPosition(3, 4, 50, 60, 70);

local third = first + second;
third.toVector3(); //Translate relative to the origin.
```

API

SlotPosition(chunkX, chunkY, posX, posY, posZ);()

Arguments

- **chunkX** (*SQInteger*) – The chunkX.
- **chunkY** (*SQInteger*) – The chunkY.
- **posX** (*SQFloat*) – The position x.
- **posY** (*SQFloat*) – The position y.

- **posZ** (*SQFloat*) – The position z.

Returns A constructed SlotPosition.

Constructs a SlotPosition.

SlotPosition(chunkX, chunkY);()

Arguments

- **chunkX** (*SQInteger*) – The chunkX.
- **chunkY** (*SQInteger*) – The chunkY.

Returns A constructed SlotPosition.

Constructs a SlotPosition, setting the position coordinates to their default of 0.

Operator+()

Returns A new SlotPosition containing the result of the addition operation on two other SlotPositions.

Constructs a new SlotPosition from the two used in the operation.

```
local third = first + second;
```

Operator-()

Returns A new SlotPosition containing the result of the subtraction operation on two other SlotPositions.

Constructs a new SlotPosition from the two used in the operation.

```
local third = first - second;
```

toVector3()

Returns An array containing the origin respecting vector.

This function does not modify the contents of the SlotPosition at all.

CHAPTER 5

Testing

5.1 Testing Overview

The avEngine employs a range of testing methods to ensure built in quality (fancy term).

There are two majour types of test that the engine supports:

- C++ unit tests
- Squirrel integration tests

These test types are explained in more detail below.

5.1.1 Unit Tests

The unit tests are c++ unit tests written to test components of the api. The practice of unit testing is pretty common. They're just tests to test simple functionality of parts of the engine.

The unit tests are written using the Google testing framework. This includes GTest and GMock.

The unit tests live in the engine repository under `avEngine/test/unit`, and have their own cmake build system, so can be built separately from the engine.

Squirrel Unit Tests

Squirrel unit tests are used to address the issues caused by exposing c++ functionality to squirrel. It is very easy to break something that previously worked when manipulating the stack, and this has the potential to destroy scripts that previously worked.

The Squirrel Unit tests, as they have been dubbed, are used to test that squirrel exposed functions work and act as they always have. They work by setting up a squirrel vm specifically for unit testing. This has the same root table setup as the actual vm. Squirrel code in the form of strings is supplied by the code. These scripts perform simple operations or checks that can return values to the c++. The Google testing framework can then be used to make assertions on these results.

It's worth noting that the entire squirrel namespace isn't tested completely here. Much of the functionality is very involved with the operating engine, and would therefore be very difficult to test as individual units. Therefore, a good deal of the namespace checks happen in the integration tests.

5.1.2 Squirrel Integration Tests

These tests are a bit involved. Integration testing is the practice of taking the individual blocks of software present in unit tests and testing them as a whole.

The integration tests themselves are actually written in the same squirrel scripting language as the scripts for the main engine, and support for them is built right into the engine. A wide range of aspects of the engine can be tested with this flexible system, and it has its own api to perform assertions and test related actions.

Usage

Code I've written to test things lives in the `avTests` repository. The structure of a test is a simple data directory similar to what would be distributed with an actual game. The only difference here is that the `avSetup.cfg` file contains an entry like this:

```
TestMode    True
TestName    SlotManagerActivatesChunk
```

This entry tells the engine that this data directory is a test, and it should start with testing mode enabled. Testing mode changes some functionality of the engine to allow for tests to be written. For instance, it enables the `_test` namespace which is where the test related script functionality lives. However, it should not be enabled unless the user is actually intending to write a test.

From there the test has access to extra information that the engine would not normally provide, to allow checking for changes in the engine.

For instance:

```
local num = _test.slotManager.getChunkListSize();
_test.assertEqual(1, num);
_test.assertTrue(true == true);
_test.assertFalse("Blah" == "BlahBlah");
```

That test would be able to check the contents of the slot manager chunk list and make assertions based on it. These methods of querying are reliant on them being implemented in c++, however the function set that the engine has will grow as more tests are created.

Running Tests

Tests are run the exact same way that the user would run any data directory.

```
./av ~/Documents/avTests/integration/SlotManagerTests/SlotManagerActivatesChunk/
↪avSetup.cfg
```

The engine has the capability to run these tests automatically, and the result of the test will be printed to the terminal when done.

Running Multiple Tests

Tests are organised into a hierarchy, which goes

- Test Projects - Large groups of tests that try and fulfil a task (integration tests, stress tests)
- Test Plans - Groups of tests that test a section of the code (SlotManagerTests)
- Test Cases - Individual tests.

The engine only allows the running of individual test cases, and it has no knowledge of the upper two layers. This is purely to keep things simple on the engine front.

Instead I have developed a python test runner application that can run entire test plans for the user, and provide the complete results at the end. This script can be found in the `avTools` repository. Information on its use can be found by running:

```
./testRunner.py --help
```

CHAPTER 6

Building

Building components of the engine is described here. This also includes best practices and tips.

6.1 Build the Documentation

This documentation is stored in the `avDocumentation` repository. It can be built with the following steps.

For a Debian based system:

```
sudo apt install python3-sphinx python3-sphinx-rtd-theme
cd avDocumentation
make html
```

The results of the build are placed in `_build`.

6.2 Build the Engine

Building the engine from source involves a number of steps. A correct environment needs to be setup so the build can complete properly. The project provides a few solutions to this.

6.2.1 Dockerised Build

A container is available to build the engine dependencies on Linux. This container is based on Ubuntu, and is generally intended to build the appimage. The produced built dependencies may not be suitable for a different host system, for instance Fedora.

This is stored in the `avBuild` repo. The user must ensure their host operating system has docker installed and can run containers.

The dependencies can be built with the following script:

```
#You only need to build the container once.
./build.sh
#Where ~/buildDir is your directory.
./start.sh ~/buildDir
```

6.2.2 Local Build

Building without the container is also possible. This is recommended if attempting to build the engine as something other than an appimage.

In the `avBuild` repository is the `linuxBuild` scripts directory.

In there `build.sh` will perform the same steps as within the docker container.

```
./linuxBuild/build.sh ~/buildDir
```

Index