LIFESPAN

INTERACTIVE MEDIA AND GAME DEVELOPMENT
COMPUTER SCIENCE

A MAJOR QUALIFYING PROJECT REPORT

SUBMITTED TO THE FACULTY OF

WORCESTER POLYTECHNIC INSTITUTE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF BACHELOR OF SCIENCE

BY:

MICHAEL GROSSFELD

COLIN OGREN

MICHAEL PELISSARI

NICK SILVIA

WILL STOCKINGER

MATT TOMSON

ADVISED BY:

PROFESSOR JOSEPH FARBROOK

PROFESSOR CHARLES RICH

PROFESSOR KEITH ZIZZA

ABSTRACT

Lifespan is an Interactive Media and Game Development Major Qualifying Project developed in the Unity game engine over the course of one year by six students. *Lifespan* is a first-person puzzle game in which the player affects objects in the environment by using a time-manipulation device. Designed with a unique spin on standard puzzle games, *Lifespan* seeks to add realism, science, and nature to the environment with new and interesting mechanics.

ACKNOWLEDGEMENTS

We would like to thank our advisors – Joseph Farbrook, Charles Rich, and Keith Zizza – for helping guide us through the development of *Lifespan* and making sure it was as good as it could be. We would also like to thank our playtesters, who gave us insight into how other people thought of our game and found a plethora of bugs, design flaws, and fun solutions that helped us make the game even better. We would also like to thank our friends and families for putting up with the long hours devoted to developing *Lifespan*.

TABLE OF CONTENTS

TABLE OF FIGURES

AUTHORSHIP

Introduction	Michael Grossfeld
Game Design	Michael Grossfeld and Matt Tomson
Genesis	Michael Grossfeld and Matt Tomson
Gameplay	Matt Tomson
Tech	
Managers	Michael Grossfeld
Control	Michael Grossfeld
RigidbodyFPSWalker	Colin Ogren
GUI	Michael Grossfeld and Matt Tomson
Level Editor	Colin Ogren
Challenges	Michael Grossfeld, Colin Ogren, and Matt Tomson
Art	
Direction	Nick Silvia and Will Stockinger
Blend Shapes	Will Stockinger
Modularity	Michael Pelissari
Character	Nick Silvia
Environment	Michael Pelissari, Nick Silvia, and Will Stockinger
GUI	Will Stockinger
Challenges	Michael Pelissari, Nick Silvia, and Will Stockinger
Sound	
Direction	Michael Pelissari and Nick Silvia
Effects	Michael Pelissari and Nick Silvia
Music	Colin Ogren
Challenges	Colin Ogren, Michael Pelissari, and Nick Silvia
Project Management	
Scope	Matt Tomson
Planning	Matt Tomson
Challenges	Michael Grossfeld and Matt Tomson
Playtesting	Will Stockinger and Matt Tomson
Post-Mortem	WallPunch Studios

1. INTRODUCTION

Lifespan began as a small idea and grew into something larger than we thought possible. Five Interactive Media and Game Development seniors and one Computer Science senior met for a series of weekly meetings in March of 2012 to flesh out a game about time travel. After seven weeks *Lifespan* had been designed. The concept of the game was then brought to three professors, who each accepted an advisory capacity on the project, and a Major Qualifying Project was born. Starting in late August 2012, we began work on what would eventually become the final product. Working with the three advisors over the course of four terms, we iteratively developed *Lifespan* in the Unity game engine.

In A Term, we worked on building the facility. Different methods of creating blend shapes were being investigated and the control systems for objects were being built. This term saw many redesigns to the facility. As such, we moved from a workflow where the facility was designed as all one piece into the modular system. A single hallway and some test chambers changed to a series of hallways and rooms with doors that blocked off access based on level flow. Every weekly iteration saw major changes to the facility level, but by the end of the term we finalized what became the pieces of the facility.

B Term saw the building of the facility and the creation of the XML Event system. Objects were being created, including some more organic models by way of blend shapes. The XML Event system was used to direct level flow for the player and proved to be invaluable for the project. The layout of the biodome was finalized, created, and then populated with objects. Object placement in the facility was getting into the end stages. Outside people could begin playtesting a barebones version of *Lifespan.* Each major piece had been blocked in.

In C Term we experienced crunch. Object placement was finalized with unique setups in every room. The biodome was populated with objects and made into solvable puzzles. The tech team fixed bugs related to the SoundManager and threading system. Art integration peaked in this term as all of the assets we had been making finally were assimilated into the game. Final textures were generated to replace temporary ones. The last few weeks of the term was spent playtesting and making small, iterative changes to better the player experience.

D Term finished the development of *Lifespan* and brought it to a few competitions. *Lifespan* appeared in the Made in MA first annual Student Showcase, where it won first place – and $250 – by popular vote. The game also appeared at WPI's booth at PAX East, where it was received quite well. *Lifespan* entered to compete in the Game Design category at the New England Undergraduate Computing Symposium, but events in Boston caused the event to be cancelled. The project was also chosen, by WPI's Game Development Club, to be WPI's candidate into the Electronic Entertainment Expo (E3) College Game Competition. The competition itself takes place after the end of the project. The term was also the formal presentation of the project on Project Presentation Day and an informal showcase at the IMGD Showfest.

The project's many artistic and technical achievements are documented in the following sections. *Lifespan* was a chance for us to showcase our skills and proved to be a learning experience for all of us.

2. Game Design

The initial step in the development of a game such as *Lifespan* is to design it thoroughly before starting and to continue to adapt to feedback by making changes. *Lifespan* was designed over D Term, 2012, in weekly meetings done in our spare time.

2.1 Genesis

The team originally formed around the idea of making a game about time travel. Once the six of us all joined the team and group meetings started taking place, the initial plan was to create a concept for a game to take to potential advisors. A puzzle game gave us the best opportunity to design a game around time travel, so it was decided that our game would be of this genre. We looked at games such as *Portal* and *Braid* to gather ideas behind the design of our game. One important point that we stated at every meeting was that while we wanted to follow in the vein of some of the popular puzzle games, differentiating our game from them was important as well. Giving our game a unique look and feel was something that the team valued highly.

After looking at other time travel games, we noticed that most powers affected the environment as a whole. One way this could be accomplished is by making our powers work on a smaller scale, such as on smaller pieces of the environment or an object-by-object basis. Another thing we noticed is that most of these games take place in a scientific and unnatural environment. The idea the team came up with is that an object-by-object mechanic could make for very interesting puzzles in a more natural environment. One setting that would showcase our mechanics well was the jungles of Brazil, in the Amazon. The jungle areas had plenty of objects in them that the player could interact with. Due to the scope of the project, the whole jungle was infeasible and we had to find another way to create a natural-like environment that the player could experience that was smaller in scope than a whole jungle. The team eventually came up with a contained environment, a biodome, where the player could feel like they are in a natural setting and still be able to create it with the given timeline and resources. We also wanted to add a place where the players could learn the mechanics of the game. Since we wanted some physical object that could manipulate the device, we decided to make it a device and that the player should steal it. The device would then be contained in a facility (Figure 1), which could transition into the man-made natural biodome (Figure 2). This gave us an interesting contrast of unnatural to natural, which would increase the interest of our game and show how our game is different than most of the same genre.

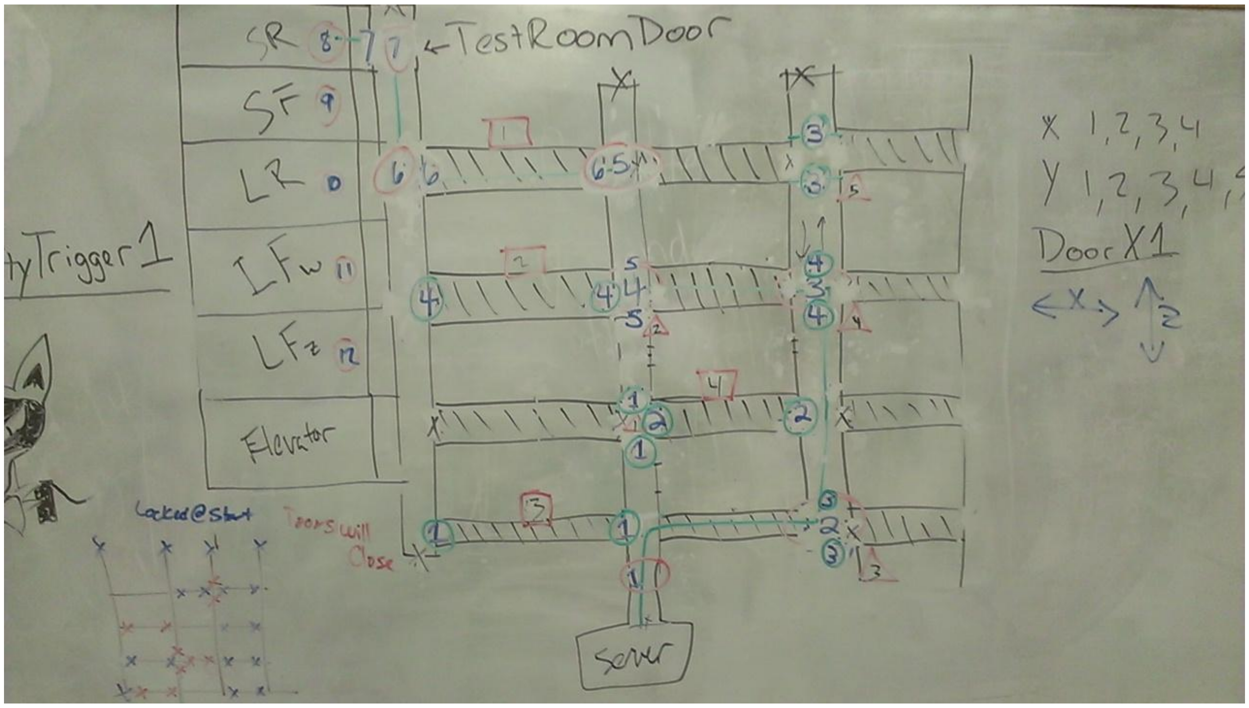


FIGURE 1: EARLY SKETCH OF THE FACILITY LEVEL

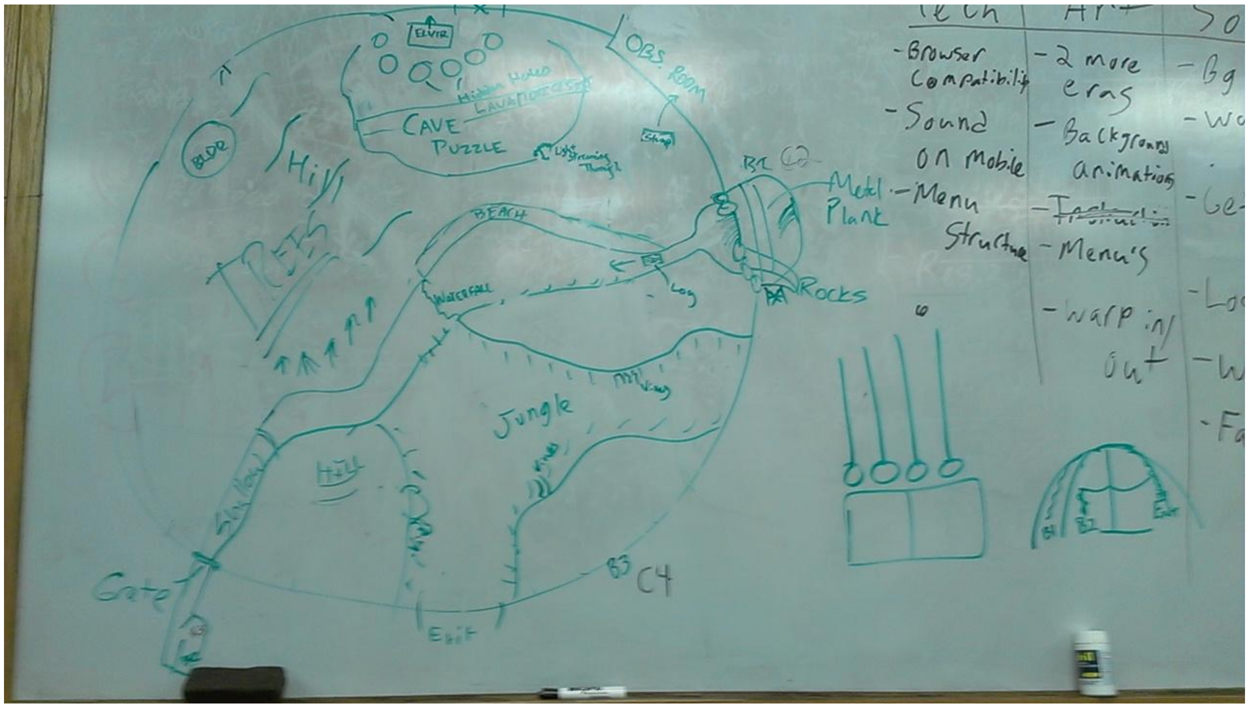


FIGURE 2: EARLY SKETCH OF THE BIODOME LEVEL

2.2 Gameplay

The gameplay behind *Lifespan* focuses on the object-by-object manipulation mechanic. Going off the idea of an object-by-object mechanic, we started to come up with lists of objects and how our mechanic could affect these objects. It became apparent that manipulating the time of an object could pertain to either the age of the object or the location of the object or both. The abilities were split into two categories because it gave the player more control of puzzle solving. We also thought that the powers would work in three directions: forward, reverse, and freeze. Location forward would move an object forward in time to where it would be, location reverse would move an object back to where it has been, and location freeze would stop an object in place. The other set of powers would affect the age of the object, which we refer to as the state. The state forward power would age an object forward in time to what it will be in the future. The state reverse power would make an object younger. And originally we thought that a state freeze power would stop an object from changing state. The issue we had with this is that we could not come up with any feasible puzzles that utilized this power, so it was cut during development.

Once we had the idea of these five powers, we wanted to come up with puzzles that the player could solve to get a grasp of these powers. One example we thought of early on was activating the state powers on a tree, moving it between three states of sapling, healthy tree, and dying tree. This was an object we used in any places in the game, therefore we thought that it would be a good object for players to test the mechanics on. Another state object we came up with was a flower, because it would be obvious as to what's happening to the flower when the player activates state powers on it. The location powers were more difficult to come up with tutorial examples for. The location reverse was to move an object back after a player saw it moving. The location forward needed to move something forward with a purpose. The location freeze needed to stop something moving. It took a long time to come up with interesting beginning puzzles that also gave the players ideas of how the biodome worked, since the biodome was the most important part of our game. We had designed several puzzles for the biodome area, such as a rock rolling into a grate to break it, freezing a log on a river so that the player could jump across it, or reversing a switch from off to on. We added puzzles like this to the tutorial section to both explain the powers, explain how to use them, and give players ideas on how to solve some puzzles that occur in the biodome.

An important part of the design of *Lifespan* is that the mechanics and gameplay were fluid, always changing based on design meetings, playtesting, or advisor suggestions. We changed the mechanics many time in the interest of making them easier to understand, more interesting to use, and more fun to the player. One design change we made over the course of the production of *Lifespan* was to change the location forward to work for objects that are not moving. Previously, they would not move if location forward was activated on a stationary object. This was changed to add velocity to a stationary object, as players expected that to happen. Another change that was made was to change the behavior of activating either location reverse or location forward on an object that was being affected by location freeze. Previously, activating the forward or reverse on a frozen object would only unfreeze the object. After the change, an object would both unfreeze and either forward or reverse based on what power was selected. This made more sense to the player.

3. TECH

The technical team behind *Lifespan* consisted of Michael Grossfeld, Colin Ogren, and Matt Tomson. Programming in C# and using MonoDevelop with Unity allowed the tech team to create numerous systems that controlled the player's interactions with *Lifespan*.

3.1 MANAGERS

In order to facilitate a faster iterative development process, we decided to create a framework which would allow us to quickly change entire aspects of level layout. The Manager system was devised as a series of event-driven systems that would call upon one another for very specific purposes. Initially, five managers were devised: SceneManager, EnvironmentManager, ScriptManager, SoundManager, and AIManager. These managers would work off an XML file which would not only be easy to read but easy for anyone on the team to edit. After some design decisions, the AIManager was folded into the EnvironmentManager.

The Manager system enabled us to change the flow of our first level, the facility, in shorter times. Generally, when triggerable events need to happen in Unity, a number of triggers need to be placed in the scene, and then for each trigger a script needs to be written for that particular action. Instead, a single script was written which would send an event to the Managers, who would then figure out what to do based off of the XML. This meant that each trigger did not need its own script, but just an instance of a single script with a specific attribute. In practice, the Manager system enabled us to redesign the facility about a dozen times with each time getting cheaper in terms of hours spent.

3.1.1 XML

In order to facilitate the creation of the Manager system, we designed our own XML schema for events. Each level works off of a unique XML file, which helped to separate the design of the levels.

Inside each XML file is an overarching array tag, called events. All events go into this array, and are read from this array. An event has a name, which is a unique identifier that is called by the Manager system. A single event is broken down into the following categories:

- Scripts
- Sounds
- Environments

Each of these categories works with a specific manager to control the flow of a level, in-game.

Scripts

The scripts tag contains lines of on-screen text that need to be displayed. A sample from the scripts tag looks like the following example:

```
<line dialogue="The State Reverse function reverses an object's state
back in time." sound="null" skin="mission" interrupt="true"
duration="7"/>
```

The dialogue variable is the text that actually needs to appear on screen. This can be of any length, though it is best if kept short. While the manager for scripted text manages long lines, the more text visible on screen at any point in time, the less of the game that is visible.

The sound variable is the name of a sound event, in this particular event, that should play when this line of text is executed. This sound event must be located in the event that contains the particular script tag or else it won’t be executed.

The skin variable determines where this line will display on screen. This is determined based off of settings in the ScriptManager. In the example above, the line will be shown in the area designated as “mission,” which is in the upper center part of the screen. This is useful for having different areas to display text at the same time.

The interrupt flag determines whether or not this line interrupts any other line in a specific skin. For instance, if the example came after a line that lasted for minutes, the example would interrupt up and immediately be displayed.

The duration variable is how long, in seconds, that the line appears. If a line needs to be displayed indefinitely, a very large number needs to be used, such as 9999999999. Duration is useful for instructions that don’t need to be visible forever.

These variables, paired with the ScriptManager, display text on screen for the player. It allows for an initial setup and fast, iterative changes from one document, as opposed to having to go into many different scripts to change the lines.

Sounds

The sounds tag contains sound events that need to be executed when the main event triggered. These sounds are uniquely named for an event. The sound tag looks like the following example:

```
<sound name="walkRightFootstep" tag=""
audioClip="Facility/Facility_Walk2" delay="0.0">
      <flags>
        <forward value="true"/>
        <volume value=".01"/>
        <pitch value="1"/>
        <pitchVariation value="0"/>
        <fadeIn value="0.0"/>
        <fadeOut value="0.0"/>
        <crossfades>
        </crossfades>
      </flags>
    </sound>
```

If it seems like there are a lot of variables here, it is because there are. Sounds are extremely complicated and have a lot of different options for each event. Not only is each sound uniquely named, but the tag variable allows it to execute on any object in game that shares that tag. This is useful for having doors slam or alarms to ring.

The audioClip variable is the filepath to the actual audio file stored in the Resources folder. Without this variable set properly, the sound won't play because the Manager system will be unable to find the correct sound file.

The delay option is to delay the sound event from triggering. This is done as a floating point number and is in seconds. If the delay is set to -1, it is not executed with the event execution, and is instead loaded for later play. However, if every sound in an event has a delay of -1, as is the case with a music track, then one of these sound events is chosen at random to begin playing.

There are a number of flags that each sound has with it. These flags can optionally be included, as they are in the example, for more precise tweaking. Most of these flags are self-explanatory, such as volume, fadeIn, fadeOut, pitch, and pitchVariation. The forward flag determines whether or not the sound will play forwards or backwards; this is useful for a game where the main mechanic is time manipulation. The remaining flag, crossfades, is a bit tricker to explain.

The crossfades flag is an array of other sound events that this particular sound plays at the end of its execution. This was done so that we could have dynamic music, where each specific loop has a list of other loops that can immediately follow it. An example of a crossfade tag looks like the following:

```
<crossfade name="hvacG01" time="1.0" />
```

This tag has a name, which is the name of a sound in this particular event. It doesn't reference a sound in a separate event, or the name of a separate event. The time flag is the length of time for a crossfade between the currently playing sound and this crossfaded sound.

A sound event can contain many crossfades, and one of them is chosen at random by the SoundManager for playback. This is the dynamic playback feature, and it allows the sound designer to create a musical flow, with biases, for specific loops to lead into other loops.

Environments

The environment tag works to trigger specific functions on objects with the given tag. Here is an example of the environment tag:

```
<environment tag="DeviceRoomDoor6" function="door" flag="close"
delay="0"/>
```

The environment event works over objects with the given tag variable. In this example, the door for the sixth device room is the object that will be told what to do. The function variable dictates which functions that object is supposed to execute, such as “door,” and these functions are executed with the given flags (“close”). This door is being told to close with a delay of zero seconds, but the delay could be set to any floating point number greater than or equal to zero.

The environment tags are executed in the order of which they appear in the XML file. Like the other tags, there can be as many of them as needed per event. The EnvironmentManager takes them in and runs them when the main event is triggered and sent to the SceneManager.

3.1.2 SceneManager

The SceneManager was seen as the manager which controlled the other managers. Events would be sent to the SceneManager, who would then send the other managers the pieces that they needed. The SceneManager made things easily by consolidating the event messages into a single script so that we didn’t have to manually send the event to each of the other managers every time.

When a script triggered an event, it sent a string as a message to the SceneManager script in the scene. The SceneManager would then lookup the name of event in the XML file, which had already been converted into easily searchable classes. Based off of the contents of the event, the SceneManager would then send the event to the EnvironmentManager, the ScriptManager, and the SoundManager.

3.1.3 EnvironmentManager

The EnvironmentManager began as two different managers, itself and the AIManager. After some design changes due to scope, the AIManager was folded into the EnvironmentManager. The EnvironmentManager was seen as a way to initiate specific actions in the scene when an event was triggered. Actions such as starting animations, opening doors, or occluding objects that didn’t need rendering were all taken care of through the EnvironmentManager.

This manager, like most of the managers, worked off of tags and message sending. An environment event had a tag specified and a function name. When the manager was given an event, it sent that function, and any flags associated with it, to every object that had the given tag. This enabled us to quickly activate multiple objects, such as the alarms in the facility. However, when it came to do more precise actions, such as opening specific doors, each object needed its own specific tag. What this meant was that at the end of the project we had upwards of two hundred tags in the system.

Had AI been implemented within *Lifespan*, the EnvironmentManager would have taken the care of the actions required for that as well. Switching specific flags to tell the hierarchical state machines to change would have been trivial for the manager to work with.

3.1.4 ScriptManager

The ScriptManager was designed with the thought of subtitles in mind. In the early designs of *Lifespan*, Jacques would talk to himself often. This would provide a stark contrast to other, similar puzzle games that had silent protagonists. With his dialogue would come subtitles, so that people would have the option to read along or read instead of listening.

As development progressed, the voiceovers for Jacques were scoped out. However, the ScriptManager was still in development for other reasons. We decided that the device would have mission text, which would inform the player of what they were supposed to be doing. Additionally, we had a console visible on screen which served as the device's diagnostic output story-wise.

The ScriptManager worked off of two lists: the name of the outlet, and the GUIText associated with that outlet. The script event in the XML would tell the ScriptManager which outlet to send the associated text and any flags that would be associated with that event, such as duration or formatting. The text would then be displayed on screen.

3.1.5 SoundManager

The SoundManager was the most difficult of the managers to get working correctly and that was because it required the most fine tuning. With sound playback in video games, timing is essential. Unity doesn't support perfect timing, nor do most game engines, because it is very difficult to do. Another requirement we wanted of the SoundManager was support for dynamic music, so that instead of one very long, looping track that the player would get bored with, the score would be comprised of different loops which would dynamically mix with one another depending on a number of factors.

Initially, the SoundManager was a simple manager that controlled all sound playback in the scene that wasn't ambience. Sound effects, collision sounds, and music were all going to just be generated and maintained by the SoundManager. These different aspects of the SoundManager proved to be more difficult than originally envisioned.

The SoundManager grew to include volume control, pitch and pitch variation, fade in and fade out times, and a modifier which adjusted which direction the sound would play (forward or backwards).

Volume control was a floating point number which went from zero to one. This allowed us some flexibility with individual sounds and then individual events associated with those sounds. If a sound file was particularly loud, we were able to adjust in the XML its volume. Different collision sounds could share the same sound file but have different volumes dependent on how strong the collision was.

Pitch and pitch variation were floating point numbers as well that were able to vary greatly in either direction. Adjusting the pitch positively larger would speed up the sound, while adjusting the sound any size in the negative direction would make the sound play in reverse. The negative aspect of pitch was never specifically intentional, as using the backward modifier would have been ideal.

Pitch variation, on the other hand, was a plus or minus modifier to the pitch variable. This number was generally small, and allowed for a range that the SoundManager could choose from. For example, if pitch variation was set to 0.5 and the pitch was set to 1.0, the possible pitches for the sound were anywhere from 0.5 to 1.5.This allowed for the sound effects in *Lifespan* to differ slightly so that the player was not hearing the same sound over and over again.

Fade in and fade out times were the length of time, in seconds, that the sound effect would fade in or fade out. This was mostly for the musical tracks, although sometimes having a sound fade in was useful.

The forward modifier value, in code, would take the pitch and set it to either negative or positive depending on what the modifier was set to. This was useful for some of the initial sound effects, but ultimately wasn't used very often.

Sound Effects

Sound effects in *Lifespan* are things such as alarm sounds, elevator whirrs, and the myriad of collision sounds that are attached to objects. These sounds can be played at specific locations or on an object itself.

The way that sound effects worked was, once again, XML based. A trigger would send an event to the SceneManager, who would in turn send that information to the SoundManager to deal with. The SoundManager would look at the data associated with the event and fire the sound. Regular sound effects worked off of GameObjects in the scene. Generally, those events were triggered on objects with the same tag.

Collision sounds were activated by a script attached to each object which called on specific collision events. When the script detected a collision, it sent the collision event to the SceneManager with the point of impact. The sound was then played at that position.

Originally, music loops in *Lifespan* were maintained and played under the same system that governed sound effects. Over time we found that more specificity was needed for music loops in particular which led to their own branch of the SoundManager.

Music

Early in the design process, music was envisioned as dynamically changing so that at no point would players hear a track loop. Some games choose to avoid these issues by having quite lengthy music tracks; others just go with dynamic music. Dynamic music in *Lifespan* consists of a number of short loops – between ten and thirty seconds generally – that have a specific flow to them which is based off of the XML. Each loop has a list of other loops that it can flow into and this is randomly selected near the end of the played loop.

To make this possible, our XML included a "crossfades" tag which contained a list of the other loops in the sound event. Included in this was a duration, which was the length of a cross fade between the two loops.

Moving the music from the original sound effect implementation proved to be quite difficult. It was necessary because of the peculiarities that differentiate music from sound effect were too specific to include in the same script.

Coroutines vs. Threads

Sound timing is a very difficult thing to do in Unity. As a developer there isn't a way to set an exact time for a sound to play. Attempts to do this often lead to sound firing at the wrong time. Unity's way of solving this problem is called Coroutines.

Coroutines are executable functions which run in pseudo-parallel with the main thread. Coroutines work by changing the function into a series of enumerators and then yielding the processor up to the main thread whenever a yield is called. If a sound needs to play in a range of times, coroutines can be used without fail. They'll play, maybe a bit out of sync, but it won't be terribly noticeable.

The main issue with coroutines is in how they execute. By yielding to the main thread, coroutines have to wait until priority comes back to them before they are allowed to continue execution. If a while loop begins execution while the coroutine is waiting, the coroutine won't be able to continue until that loop is complete. There is a high probability that this delay will cause something that needed to be played in five seconds to instead be played in forty. For sound effects that just needed to play once off there wasn't a need to use a waiting coroutine. The music, however, we wanted to crossfade into one another and to play as soon as the previous one was finished. Delays of forty seconds led to lulls in the music that sounded disjointed and wrong.

A new system was devised for playing music correctly. This system used C# Mono framework's threads, which are not supported in Unity. Threads run slightly differently than coroutines. They run in parallel to the main thread, as opposed to the pseudo-parallel way that coroutines work. Threads in Unity have an issue with the Unity API which is not thread safe. This means that no calls to the Unity system can occur in self-made threads – they must occur within Unity's main thread. This severely limits the usability of threads; if a thread cannot call the Unity API there isn't much it can do.

However, threads can be used for more precise timing. Because threads run in parallel, they can keep a running watch to time the loops for music. In the new system, a thread is used for each music track that is attached to the player. The thread counts for 80% of the loop's length, and then queues up, for the main Unity thread, the job of adding the new loop into the track. This created seamless music switchovers which allowed us to have the music dynamically play how we wanted it to.

AudioClipExtended

In order to facilitate the creation of music, a new script was composed that would refactor the code from sound effects and make it more focused on music. The AudioClipExtended class took the threads out of the sound effects and put them in their own class so that there weren't hundreds of possible threads being created every frame.

AudioClipExtended also utilizes the AudioSource and AudioClip classes to their fullest potential. AudioClips have the ability to have data, an array of floating point numbers, retrieved from a sound or set from another sound. This allowed us to dynamically load sounds and manipulate them directly, which meant that we could fade sounds in and out, and also crossfade sounds with precision. Having complete control over the sound's data allowed us to do volume manipulation for music loops as well, while keeping it separate from the volume for sound effects.

3.2 Control

In order for objects in the environment to be controlled, a single script was created called ObjectControl. This script took care of morphing the blend shapes and for affecting the physics of an object. Very quickly this script became monolithic and difficult to edit, so it was refactored into two different scripts: State Control and Location Control.

State Control worked by telling the Morph Blend Targets script to slightly update the blend shapes according to specific values. When a few settings were chosen in the State Control script, the update loop would inform the Morph Blend Target script to update which in turn caused the model to update. In order for updating physics models to be implemented, these small changes were then set at the mesh collider for the object so that the collider would update with the model in real time. State Control had three separate states: Reverse, Normal, and Forward. Objects started in their Normal state and were able to change to Reverse or Forward from there. Once there the appropriate ability could move them into one of the other states.

Location Control kept constant track of an object's previous positions and rotations, kept as a Dictionary of Vector3's and Quaternions, respectively. When the player activated the Location Reverse ability, the Location Control script would read the set of previous positions and rotations and forcibly move the object to them over a period of time. By storing just the positions and rotations, the Location Control was able to avoid memory leaks, which even after forty minutes of gameplay never became an issue. In order for Location Forward to work properly, the Location Control script increased the velocity of the object's Rigidbody until it reached a maximum velocity – set for all objects across the scene. Because of the increase in speed, many objects found their way moving through colliders and outside of the scene's boundaries. To fix this, any Rigidbodies moving fast enough to go through a collider was set to the Continuous Dynamic detection mode, which in conjunction to the maximum velocity cap, saw most of the objects remaining within the scene. Location Freeze proved to be the easiest of the Location abilities to control, as it required locking the Rigidbody's constraints down so that it couldn't move, increasing the Rigidbody's mass so that it can't be altered in any way, and make the Rigidbody kinematic so that no physics act on it at all. All of these changes to the Rigidbody enabled the player to keep an object stationary while still allowing them to interact with it by jumping on it.

The two separate scripts enabled us to have a very modular system when it came to controllable objects – if an object needed to be morphed, State Control was applied and Location Control was not. If an object needed both, both were placed on it. It gave us more freedom and simplified the procedure for getting an object from model to in-game.

3.3 RigidbodyFPSWalker

The RigidbodyFPSWalker (RFW) script controls all controlled movements and most of the physics for the player. The player's collider is a cylinder with a rigidbody component that allows it to interact with other Rigidbody objects according to the laws of physics. This also means that small errors can affect the player in big ways, so parts of the script contains workarounds for these issues. The following sections document different aspects of the RFW script. Note however, these features and fixes are tightly integrated and do not work in isolation of each other.

3.3.1 Walking and Running

Walking starts with acquiring a key input from WASD keys. This key is mapped to a vector indicating the direction of travel. The RFW script can combine multiple inputs, allowing the player to walk diagonally. These inputs are normalized and then multiplied by a velocity. This final vector is applied to the player Rigidbody as a velocity, independent of the rigidbody mass, allowing for complete control over the player’s speed.

3.3.2 Jumping

Jumping looks for the spacebar key input. Acquiring this key applies an upwards velocity vector to the player. The magnitude of this player is two times the gravity strength times the jump height. This vector is only applied when the player is grounded so the velocity is only applied once. Unity physics takes over for bringing the player downwards.

3.3.3 Head Bobbing

For added realism, we decided to add head bobbing. How this worked was by slightly moving the camera when the player was moving down and back up. This followed a three part curve where the player would spend an equal amount of time at the top of the curve, going down and going back to the top. When we testing this, we found that it was too much and took something out of the game, so we ultimately scrapped this functionality.

3.3.4 Vine Climbing

The initial incarnation of vine climbing was simple. When the player walks into a trigger with a script attached for vine climbing, the script sends a signal to the RFW script to activate the vine climbing state. Pressing spacebar in this state added a constant velocity change to the player’s upwards direction.

This approach did not satisfy the producers however, as player movement does not move parallel to vines on a slope. The current implementation has a GameObject at the top of a vine climb trigger. Pressing the forward key in the vine trigger causes the player to move towards this GameObject which inevitably pulls the player up the vines. The difficulty in this implementation is that triggers for vines and the target GameObjects require careful placements so the player will not start too far away from the vines, and so the player can make it all the way up the vines without climbing far above the vines.

3.3.5 Tree Tossing

Tree tossing is the last phase of tree climbing where the player is thrown into the air as a result of the tree growing underneath them and forcing the player upwards. While other objects can simply act on a force applied to them, the player has positional constraints and is kinematic, so the player does not react to outside forces. Instead a signal is sent to the RFW script which sets a state to jump with a multiplier so the player jumps extra high.

3.4 GUI

For the player to have a better understanding of the world, we needed a graphical user interface (GUI). While having images on screen is fine, we needed our GUI to be a bit more dynamic with layout control. While the main heads up display (HUD) is a single image that is scaled based on the application's window size, the other objects in the GUI needed to be controlled with more precision than just a scaling factor.

3.4.1 Reticle

One part of our GUI is the reticle. The reticle works by first getting the object that the player ability script is targeting. It then gets the bounds of the collider of the object. The next step is to find the 2-dimensional screen coordinates of the top-left and bottom-right points on the object that are on the plane perpendicular. These are the targets of the reticle, which surrounds the object. The reticle animates, so it must step-by-step move towards the bounds while targeting the object, stopping when it reaches the corners. If the player locks on to an object, the reticle automatically snaps to the corners, even if it is not at these bounds. When the player has an object locked on, the cursor moves to stay with the bounds when the player looks around the screen. When the center of the locked object goes off-screen, an arrow replaces the cursor, either an up arrow when the center is above the bounds of the screen, a down arrow when the object is below the bounds, a right arrow when the object is to the right, and a left arrow when the object is to the left. If there is no longer an object being targeted by the reticle, the reticle animates back to the default center of the screen. The special case is when the player unlocks from an object and is no longer looking at any object. In this case, the reticle snaps back to the default position. The reason for this is that when the reticle is off-screen, it tends to throw errors when animating back to the center. The fix for this was automatically hard setting the reticle back to the center.

3.4.2 Power Icons

The power icons are an integral part of informing the player as to which of the abilities are available at any given time. Each icon is an individual image loaded into a GUITexture in Unity, which used normalized values for position – this is more useful than absolute positioning, because it scales with different resolutions. Each power icon is then placed in order along the bottom, using the relative size of itself and the knowledge of ordering to make sure that each icon is evenly spaced and centered in the tray.

When the player is locked onto an object, the icons that represent available abilities are turned into the "activate" state, which is more lit up than the "inactive" state, which is faded. This functionality was added after players reported that they were unsure which abilities they were able to use on an object.

3.4.3 Object Camera

The object camera in the lower right corner of our HUD provided visual feedback for the player as to which object he or she currently had selected. This was done because players can lock onto an object and then lose sight of it – with the object camera, the player always knew what object was being affected.

The object camera works by separating the object into a special layer that no other objects are on. The camera then renders only this layer, and displays it on screen, scaled to fit within the region we designated. In addition, the camera rotates around the object slowly, so that all facets of the object are visible.

In the same region as the object camera we also included an indicator which informed the player whether he or she had locked onto the object. This was done with a simple closed lock/open lock image, which was then grayed out if no objects were selected. Along with this indicator is another power icon which shows the player which type of manipulation they are currently using on the object. If an object is affected by Location Freeze, the power icon by the object camera would show the freeze icon.

3.4.4 Menu

The menu system is comprised of several components. The first is the splash screen. The background is the *Lifespan* logo. There are 5 buttons placed in front of this, of which the third button is placed in the center and there are two buttons evenly spaced on either side of it. The first button takes the player to the first level of the game. The second button takes the player to a screen of a picture of the controls made by the artist. The third button brings up a different menu where the player can change the volume options of the music, the sound effects, or both, as well as a link to the development website. The next button plays a movie of the credits. The last button quits the game.

The pause menu contained in-game works similarly and contains many of the same options. From here, the player can resume the game, go to an identical sound options page with a website link, restart the game at the start of the first level, and quit the game and go back to the main menu. The pause functionality works by setting the timestep to zero, disabling all of the player functionality and turning off some of the GUI elements.

3.5 Level Editor

The Level Editor is a set of scripts that function as a Unity add-on. The intention of the Level Editor was to accelerate the building of game levels by allowing level designers to lock the geometric positions of objects relative to other objects. This tool was designed and implemented in fulfilment of a Computer Science requirement for the MQP.

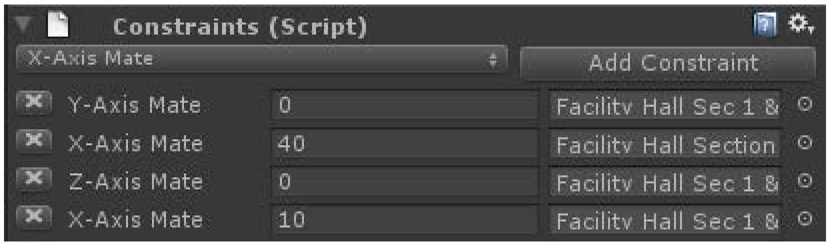
3.5.1 Development Cycle

Designing the Level Editor required learning the Unity interface and understanding the features that would help level designers to build levels more quickly. Part of this came from first-hand experience with Unity, and the other part came from consulting with the team. Requirements were then collected into the first draft of the *Level Editor Design Document*. This document was submitted this to the rest of the team for feedback and review. User stories followed along with assigning values using the unit of the jigsaw puzzle piece. As the scripts are written in C#, NUNIT testing for full Test-Driven Development was investigated. However, scripts cannot run without the Unity engine compiling them, so NUNIT was infeasible in this case. Since Test-Driven Development was infeasible and having both a design document and user stories was redundant, the decision was made to forgo user stories and stick to implementing features from the design document while testing features manually in Unity.

As implementing the design document progressed, certain challenges needed to be addressed, and the result of solving these challenges resulted in a second draft of the *Level Editor Design Document*. This draft can be found in Appendix C. Further challenges during implementation and changes in requirements resulted in more changes as the Level Editor took form.

3.5.2 Design Requirements

The Level Editor exists as a component that level designers can add to a Unity GameObject. This component allows level designers to connect the GameObject to other GameObjects by adding a new constraint and dragging the objects name into the corresponding field. The second object will receive a Constraints component (Figure 3) of its own with the first GameObject listed under the same type of constraint. Constrained objects will maintain the specified distance between them on the given axis.



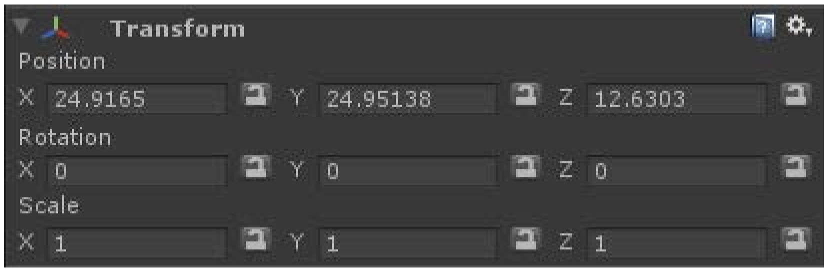
Constraints (Script)

X-Axis Mate | Add Constraint

Y-Axis Mate	0	Facility Hall Sec 1 &
X-Axis Mate	40	Facility Hall Section
Z-Axis Mate	0	Facility Hall Sec 1 &
X-Axis Mate	10	Facility Hall Sec 1 &

FIGURE 3: THE CONSTRAINTS COMPONENT HANDLES RELATIONSHIPS BETWEEN GAMEOBJECT.

Another feature of the Level Editor lies in a GameObject's Transform component (Figure 4). When the Constraints component is added to a GameObject, lock buttons appear next to each dimension. These buttons toggle whether the corresponding dimension can change. For example, locking translation on the x-axis prevents level designers from moving the object along the x-axis. Both transformation locks and constraints save with the scene.



Transform

	X	Y	Z
Position	24.9165	24.95138	12.6303
Rotation	0	0	0
Scale	1	1	1

FIGURE 4: THE NEW TRANSFORM COMPONENT CAN LOCK A GAMEOBJECT ON ITS AXIS BY POSITION, ROTATION, AND SCALE.

3.6 Challenges

SoundManager

The SoundManager had many different implementation challenges associated with its development. Early on in the process, the SoundManager played all of the different sound effects and music using Coroutines. As outlined earlier, Coroutines don't work like threads. Due to their nature, they can cause delays in the activation of sounds in-game. Switching over this implementation to a new version with threads made our music playback work, but it wasn't without its headaches.

Because Unity isn't thread-safe, all API calls are banned from user-generated threads. This meant that the threads that the AudioClipExtended class would create couldn't actually maintain the new data but instead had to rely on the main thread to do so. These calls proved to be difficult to execute properly and often caused Unity to crash. A bug showed up where on the second time of playing the game in the Unity Editor, the editor would crash and require us to restart Unity which would cut into development time. Eventually the bugs were traced back to threads not being joined on application quit, which is handled awkwardly by the editor. When those bugs were fixed, Unity stopped crashing and the music within the levels played flawlessly.

Another issue that arose during development was the handling of the sound file data. AudioClips have a feature which lets a developer to get and set the actual data behind the sounds. In order to affect music loops for crossfades or pitch variations, this direct manipulation of the sound data was essential. The data proved to be problematic, as the sound data is just an array of floats. Without know exactly how this data translated into sound there was a long period of time where all attempts at changing the data resulted with sections of white noise. Eventually the issues were worked out and the data alteration also was working without fail.

3.6.1 RigidbodyFPSWalker

Hill Sliding

As the player's collider is a capsule, the bottom is rounded. This causes gravity to pull the player down slopes when the player is not walking around. To prevent this sliding, Rigidbody constraints lock down the player when no keys are being pressed. When keys are pressed, the constraints are disabled, and the player can move.

Collisions on a Stationary Player

While locking the player's position prevents him from sliding down a slope, it also prevents moving objects from pushing the player. To combat this issue, the RFW script checks collisions with the player and when it finds one, it checks to see if it's above a certain height on the player. If the collision point is high enough, it is safe to assume that the colliding object is not the floor. Position constraints on the player are disabled, and the force of the object is applied to the player.

Wall Debouncing

Wall Debouncing is a term coined to describe a fix on the way Unity handles collisions. Unity detects a collision when the two colliders overlap. The engine registers the collision and snaps the two overlapping colliders apart so they can collide again. This system works until rigidbody player control gets involved. Holding down the forward key will move the player until he hits a collider. Unity will detect the collision and then pry the two colliders apart. If the player keeps moving forward, he will again collide and Unity will repeat the process. This creates an oscillation on the player as he moves in and out of the collider.

To fix this, the RFW script detects the initial collision and traces a vector between the point of collision and the player's center axis. This vector is rotated ninety degrees so to lie tangent to the collision point. The player is wrapped in a trigger which detects the next collision before it happens. The RFW script responds to this forecasted collision by constraining all movement to lie along this tangent vector. This forces the player to move parallel to the surface, preventing it from colliding with it again only when the initial movement vector points into the surface. In this way, the player can still move away from the surface without the tangent vector affecting movement. Jumping also works outside of this system as the player cannot continuously move into the wall while in the air.

3.6.2 Reticle

The reticle proved to be an interesting undertaking for the tech team. The major issue that occurred was handling boundary conditions for the reticle when it went off the screen. Unity threw one particular issue where when one of the two reticle pieces would disappear sometimes when it was near the edge of the screen. As this bug was not reproducible, we had to find a workaround for this bug. What happened is that when doing some of the math related to getting the screen coordinates of the reticle piece, the position would become NaN and would break. This was solved by checking before the reticle was drawn that the coordinates are numbers. If not, it set them to numbers. This solved the issue for the most part, except for a very rare occurrence of this same error.

3.6.3 Level Editor

Requirements

Sourcing requirements from the team was difficult for a variety of reasons, the largest being that consulting occurred during the summer. Team members lacked motivation to consider what a Level Editor would need to effectively accelerate level design, and as the team members lived in different locations, all communication was constrained to email, the game blog, and Google Plus Hangout sessions. We speculate that another reason was that team members familiar with Unity's workflow could not see the need for a level design acceleration tool.

Another issue with requirements came up very late in the implementation of the Level Editor. Miscommunication over terminology between team members resulted in an incongruent understanding of the requirements. Some members thought that GameObjects could be mated in local coordinates when the design specified that a GameObject could rotate around another GameObject in world coordinates. This misunderstanding was not cleared up until the final week of implementation, but this turned out to not matter as time did not permit its implementation.

Implementation

The Level Editor had many technical challenges during implementation, the largest of which are saving constraints, nesting prefabs, and computation of many constraints and many GameObjects.

For saving constraints, the Level Editor initially implemented db4o, but it moved to Unity's built-in serialization so Constraint components would save with the scene file. Unity cannot distinguish between GameObjects with the same name however, so ID tags were added to each GameObject to ensure that each one could be uniquely identified. The scene also needed a trigger to reload the constraints whenever the scripts recompiled or the scene loaded, which Unity managed to provided methods for through the API.

Another problem arose with Unity prefabs. Unity cannot nest prefabs. While Unity can place prefabs inside prefabs, the inner prefab connection is lost and the prefab becomes a GameObject hierarchy. This design breaks the grouping concept for prefabs, but the feature is still in the final implementation as there is no way around this, and the feature works in all other instances.

The final problem with the Level Editor was computation. In a small scene with few GameObjects and constraints, the constraint tree is small and depth-first search moves quickly. In a large scene however, constraint computation slows to a crawl. In addition, engine errors appear with regards to the customized Transform component on constrained GameObjects. To solve these problems, the dimension locks were moved into the constraint components and the Unity Transform component was restored to its default. Updating constraints were also delayed until the level designer finished moving GameObjects, eliminating continuous computation lag. This solved speed issues to a degree, but not entirely. Unfortunately, time did not permit finishing optimization for the Level Editor.

4. ART

Lifespan's art team was staffed by Michael Pelissari, Nick Silvia and Will Stockinger. They were responsible for all of the visual aspects of the game, including models, textures, animations, lighting, and many other aspects that went into defining the visual experience of *Lifespan*.

4.1 DIRECTION

When designing Lifespan, the main philosophy was to have an art style and feel that tied the world, the character, and the gameplay together as a single entity. The three pillars of the art style are realism, science, and nature. Although science and nature are opposing styles, the contrast was key in getting the feel of our environment across to the player. These pillars define *Lifespan* as well, using science to manipulate nature in a realistic looking world.

Players needed to see the state changed objects morph in front of their eyes. Immediate feedback makes it easier for the player to see how powers affect each object in real time. To approach this issue we used blend shapes to change the environment in real time along with their physics. Changing and morphing nature allows us to mix our art style and the mechanics of our game together. Synergizing gameplay and artstyle allows us to create a more immersive experience for the player and highlight our pillars of design.

The character needed to be interesting and relatable and represented as such through the art and animations that come with it. In order to do this we exaggerated pieces of the character and animations to make them easily understood and so they can resonate easier with players. This tied together with the device and the HUD once again create this contrast with the environment and the assets of the game, this time allowing them to easily pop.

4.2 BLEND SHAPES

Blend Shapes were essential to the main mechanics of *Lifespan*. When the player needs to age or de-age an object, the model needs to change to reflect that. In order to get a smooth transition between states of an object, we employed Blend Shapes on every dynamic object.

4.2.1 PROCEDURE

To make models Blend Shape ready (or "dynamic"), we started by modeling one state of the model. After making this model, and accounting for any requirements that other states may have, we duplicated the state we already modeled and altered the duplicates only by moving vertices (Figure 5). The team took into consideration the dimensions and pivot points of each state of the model, because an off-center pivot or strange transformations on one state of a blend shape could cause odd animations.

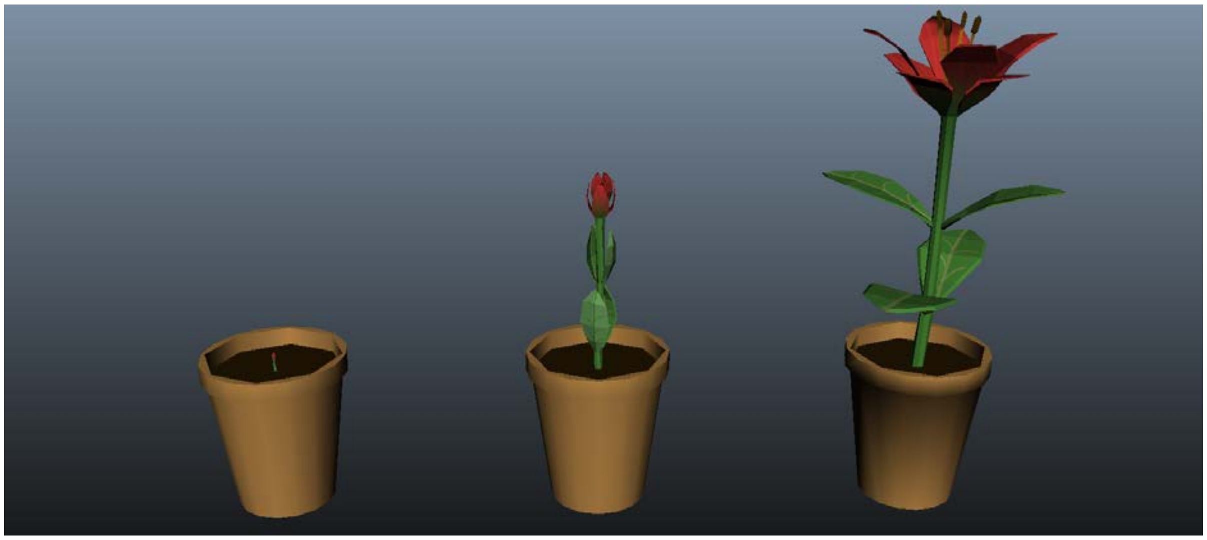


FIGURE 5: THE FLOWER DYNAMIC MODEL, SHOWN WITH ITS STATES LEFT TO RIGHT: REVERSE (YOUNGEST), NORMAL (MIDDLE), FORWARD (OLDEST)

Once we had made each stage of the model, we used Maya's blend shapes tool to test the animation between states, and then used a Maya Embedded Language (MEL) script that converted the blend shape into a Unity accessible blend shape. After the MEL scripts were run, the blend shape was exported out as an FBX model and then imported into Unity. It was necessary that when being imported into Unity, the normal smoothing was calculated and turned up to 180 because without that option set the vertex counts between the different blend shapes were off. After being imported, the blend shape scripts are attached to the model and the State Control scripts added as well. The blend shape would then be ready for use within Unity and the game.

4.2.2 Trees

Making dynamic trees requires two additional steps compared to other dynamic objects (Figure 6). Leaves need to be modeled separately in order to make a natural looking tree that morphs easily and well. The leaf model is created in the same way as any other dynamic model, and then is brought into Unity and set as a child object to the tree trunk so they both deform at the same time (Figure 7). This is also present with the vine dynamic model, which has leaves and smaller connecting vines added after the main model.

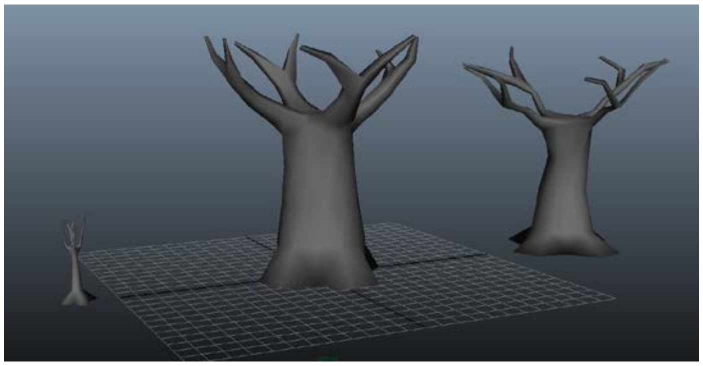


FIGURE 6: AN EARLY ATTEMPT AT MAKING A REALISTIC LOOKING TREE DID NOT GO WELL.

Second, trees that are state forwarded into old trees fall down and leave a stump behind for the player to interact with. To get this to work properly, a second tree dynamic model needs to be made by taking the forward state model and turning it into a stump. In Unity, the original tree model is replaced with a stump when the player knocks it over, allowing them to use both the knocked over tree and the newly created stump in their puzzle solving.

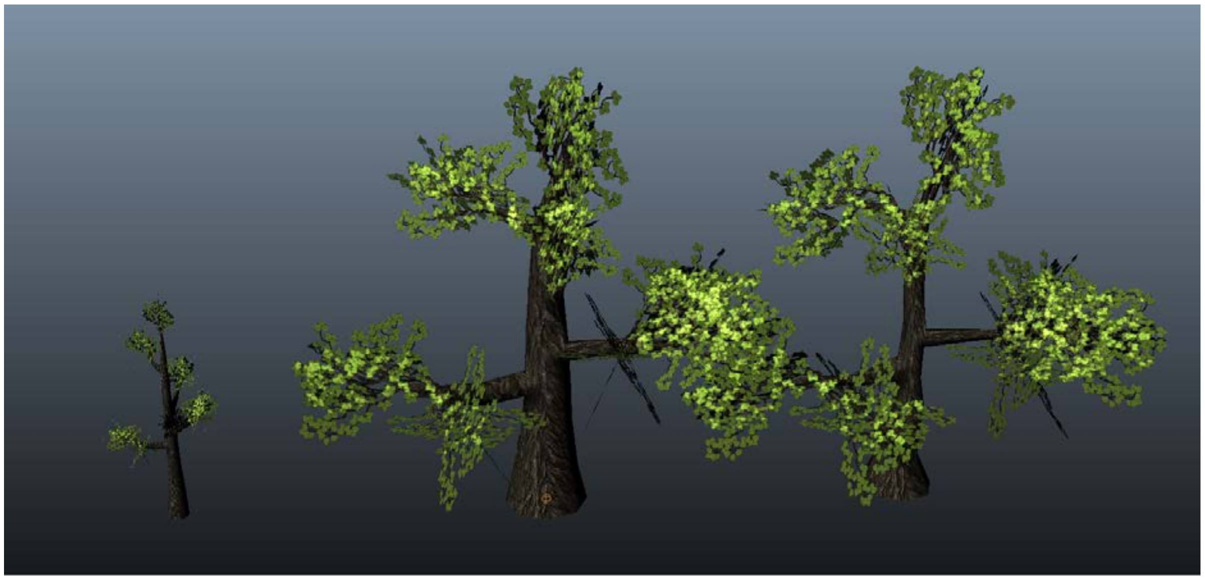


FIGURE 7: OUR FINALIZED DYNAMIC TREES.

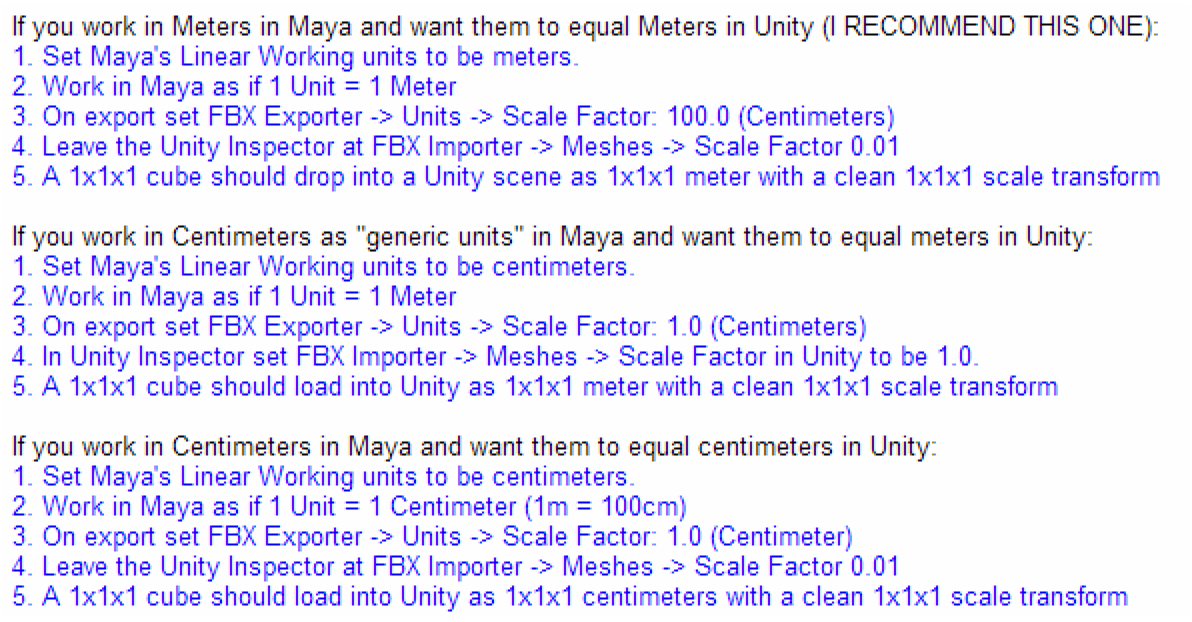
4.2.3 Glass to Sand

The glass to sand dynamic model had an additional piece to give it another dimension and make it as real as possible. In Unity, a particle effect (made out of the sand texture) was made to be played as the glass is morphed, which helps give the feeling that the sand is made up of little grains rather than just one single block.

Another thing that the team did, in order to get the glass to look like glass and the sand to look like sand, was blending the textures as the model morphed. Our tech team used a custom-written shader to crossfade between two textures in order to get the transition between states of the object to look good.

4.3 Modularity

Scale is a very important part of the game development process. Proper scaling helped different individuals to work on separate GameObjects and environments. This helped make everything in the scene to be the same size so that the player would feel as if it was real life. Since the entire team used Maya, the FBX exporting options to Unity were fairly simple. Depending on the artist, the export and Maya settings were one of these three (Figure 8):


If you work in Meters in Maya and want them to equal Meters in Unity (I RECOMMEND THIS ONE):
1. Set Maya's Linear Working units to be meters.
2. Work in Maya as if 1 Unit = 1 Meter
3. On export set FBX Exporter -> Units -> Scale Factor: 100.0 (Centimeters)
4. Leave the Unity Inspector at FBX Importer -> Meshes -> Scale Factor 0.01
5. A 1x1x1 cube should drop into a Unity scene as 1x1x1 meter with a clean 1x1x1 scale transform

If you work in Centimeters as "generic units" in Maya and want them to equal meters in Unity:
1. Set Maya's Linear Working units to be centimeters.
2. Work in Maya as if 1 Unit = 1 Meter
3. On export set FBX Exporter -> Units -> Scale Factor: 1.0 (Centimeters)
4. In Unity Inspector set FBX Importer -> Meshes -> Scale Factor in Unity to be 1.0.
5. A 1x1x1 cube should load into Unity as 1x1x1 meter with a clean 1x1x1 scale transform

If you work in Centimeters in Maya and want them to equal centimeters in Unity:
1. Set Maya's Linear Working units to be centimeters.
2. Work in Maya as if 1 Unit = 1 Centimeter (1m = 100cm)
3. On export set FBX Exporter -> Units -> Scale Factor: 1.0 (Centimeter)
4. Leave the Unity Inspector at FBX Importer -> Meshes -> Scale Factor 0.01
5. A 1x1x1 cube should load into Unity as 1x1x1 centimeters with a clean 1x1x1 scale transform

FIGURE 8: PROPER PROCEDURE FOR CONVERTING UNITS BETWEEN AUTODESK MAYA AND UNITY3D

Gridding is essential in creating a modular environment. Gridding allows a model to be designed to snap to a grid within the modeling environment or game engine. For a game modeler the “rule of two” is a golden law. All pieces should be within the range of:

- 2 x 2
- 4 x 4
- 4 x 6

The pieces should be set to multiples of two and even numbers. The multiples can be in meters or centimeters. The only real difference is the decimal point and import scale number. When we modeled the environment pieces it was important to keep the boundaries of the model polygons with in the grid. This was so that other pieces could line up easily and stop tiny gaps from appearing when the pieces were tiled.

With the creation of modular game pieces came the task of creating textures. Since the game assets for the environment were meant to seamlessly fit together, tiling textures are a necessity (Figure 9). These textures are hard to create because they cannot contain seams, thus allowing different pieces to be joined together and look like they would exist in the real world. The other benefit about tiling textures is that they made UV mapping GameObjects simple. The artists did not have to worry about seams and overlapping unwanted lines nearly as much as textures that do not tile (Figure 10).

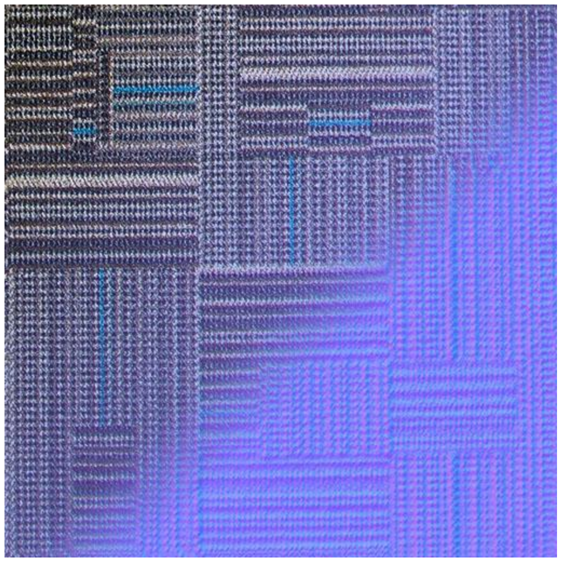


FIGURE 9: EXAMPLE TILING TEXTURE (DIFFUSE/BUMP)

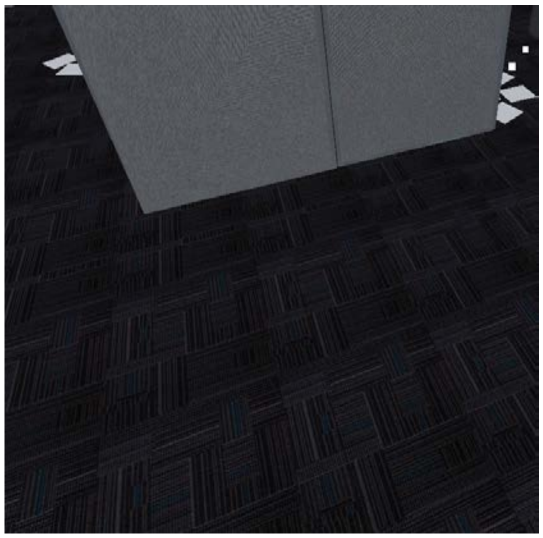


FIGURE 10: TILED IN GAME MANY TIMES

Modeling the game environment was a very long and iterative process. We started prototyping the game over the summer of 2012. The designs of our level were greatly affected by this modular system. Every iteration we refined the models and our level concepts. When our environment was completed we put different things into prefabs, which allowed us to change the layout of our level from feedback received during playtesting. Early on, the process to redesign the levels took us hours, but in the end it only took us 20 minutes to perform the redesign.

4.4 Character

Jacques is the main character of Lifespan, a suave French-Canadian super thief that has broken into the N.E. Corporation to steal the device. Jacques motivations are purely for money and personal gain; he has no concerns for what happens after each job is completed as long as he gets paid.

The design of this character was heavily influenced by secret agents, such as James Bond or Archer. Many of his features were very much exaggerated in the original concepts, though the final design of the character was far more grounded in reality. This idea of realism stretched throughout the rest of the design of assets ingame and began to define the previously mentioned three pillars of design.

4.4.1 Design

Jacques character design is very much inspired by that of Archer from the show of the same name. He dons a black turtleneck sweater and black cargo pants along with his slicked-back black hair and combat boots. Although the player can only see the arms of the character, the original design concept had the entire outfit in mind to give the character a bit more style. Using what we had, such as the gloves and the sweater, we tried to define the character in a minimalistic way.

The device is divided into three separate pieces, each one serving a unique purpose and function. The device's backpack houses the supercomputer which calculates the data for each object it manipulates based on pre-existing data it has recorded. The supercomputer has been condensed into a backpack design to make it easier for subjects to carry around, allowing testing to leave the confines of the labs.

The visor has a built in heads up display allowing subjects to see what power they are using currently and what objects they can manipulate using the device. The main purpose of this piece of the device is helping the user to easily manipulate objects and give them an understanding of what object they are affecting and how.

The glove allows the user to interact and select which power they would like to use, along with allowing them to target objects using the projector on the base of the palm. Opening the palm and pointing it towards an object allows the user to manipulate the object by using the currently selected power. All of the devices pieces work synergistically to allow users to manipulate the different states and locations in easy way.

4.4.2 Model

The model for Jacques is of just the arms. It is rigged with the entire body in mind, however. From a design standpoint, the arms needed to show something that represented the character well. Only modeling the arms was a design choice to save on polycount, as there were over three million triangles in the biodome alone. Any place we could cut polygons we did. Displayed on both arms is Jacques' trademark black turtleneck sweater while on the left arm he is wearing a brown leather glove, showing his style and taste. On his right arm is the device itself, with the turtleneck sweater arm rolled up to accommodate the size of the device.

Animation

Creating a full body rig for the model allowed animations to feel natural and professional. This rig incorporated a full range of movement to the arms and allowed the animations to look like they are coming from a larger model. An example where the full body rig was useful is when the arms first come up after obtaining the device. It was much easier to have the recoil animated when there was control of more than just the arms. Animations where the whole body is involved in the action, like most movement, was far easier when a full body rig can be utilized (Figure 11).

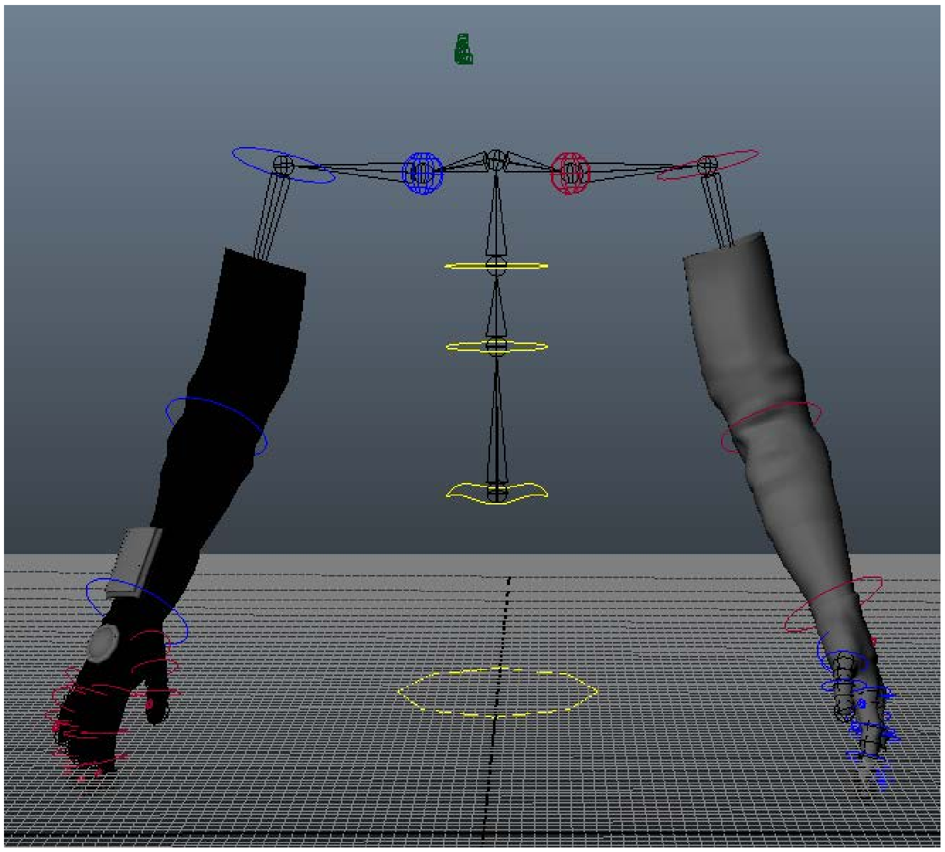


FIGURE 11: THE FULL BODY CHARACTER RIG.

The animations themselves were based on many different references. The biggest video game influence for the animations was the *Half-Life* series. Along with using different videos and games as reference for how an animation looks, the animators used their own arms and bodies to simulate the animations. The idle animations are almost entirely based on the motions of the head animator. Using many references allowed the animations to mimic and at times exaggerate real life to make them feel realistic to the player.

4.5 Environment

The art style of Lifespan is founded on three distinct pillars: realism, science, and nature. Each pillar defines the art, sound and feel of *Lifespan*. These pillars represent a goal for the art team to create something new and interesting in a market saturated with puzzle games of similar styles.

Realism is the focus of our art assets; the assets needed to feel like they were grounded in reality. The art style aimed for immersion and helped players suspend their disbelief while engaging themselves in the experience of *Lifespan*. This decision was inspired by gorgeous games such as *Crysis* – similar artistic choices can make games far more appealing to players in the current generation of games.

Nature and science work together in contrast to give our game a unique style and feel. Science is the idea of clean-cut and artificial looking assets displayed mostly in our facility level. These assets are scattered through our biodome as well, creating a contrasting look that sets it apart from other games in the same genre. While nature influences the organic and vibrant assets, such as our trees and flowers inside the biodome, these differing facets of style, with our placements, create a nice contrast that sets us apart from one of our biggest inspirations, *Portal*. The natural assets together with an overarching feeling of science – a giant glass dome and walls – remind the players that while they might be more free, they are still contained within the N.E. Corporation. Creating a unique and appealing art style was a key part of our development to separate our game from the other first person puzzle games on the market.

Each of these pillars are represented in the environments of our two levels: the facility which is more scientific and oppressive, and our biodome, which is grounded more in nature and freeing. Both levels are unique and are reflected in the art, sound and music.

4.5.1 Facility

The game starts in the Facility level, where players obtain the device, get acclimated to it, and make their way out using its powers for the first time. Many of the assets in the Facility were designed primarily with science in mind (Figure 12).

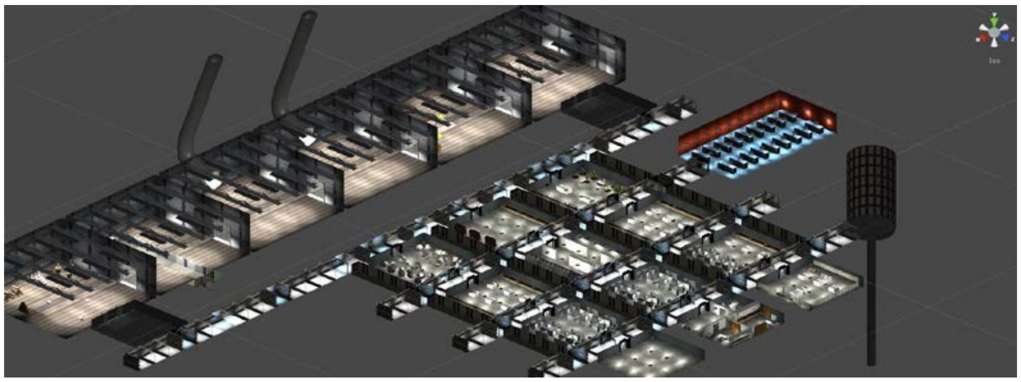


FIGURE 12: AN EXTERIOR VIEW OF THE MAIN AREAS OF THE FACILITY.

Test Rooms

Our test chambers are a big part of the facility level. They are the rooms that teach the player about our five unique device powers. The rooms were heavily modeled after the rooms in *Portal*. We wanted to give the test rooms a dark and sterile feel. This was to give the player a sense that he is being observed and used as a rat in a maze.

Office Rooms

Throughout the facility level there are various rooms that emulate a typical workplace. These rooms are filled with cubicles or are high executive suites, which are furnished with standard office furniture. We designed these rooms to look like the employees in them got up and left in a hurry.

Lab Rooms

The lab rooms were the rooms that held all the extra resources that would go into the testing rooms. This included items such as trees, steel grates, and boxes. These rooms have a feeling of sterility, use, and storage.

Hallways

The facility is laid out in a typical grid pattern. In between each room there are two different kinds of hallways. The main hallways, which are bright and colorful, are made for heavy traffic (Figure 13). Connecting the main hallways are smaller service corridors which have low emergency lighting, making them dark and dank. Our hallways all look similar to give off a conformist feeling, but have various pieces of clutter to keep players from getting lost. The elevator is an open lift in a shaft with an up and down button to traverse the many floors of the facility.

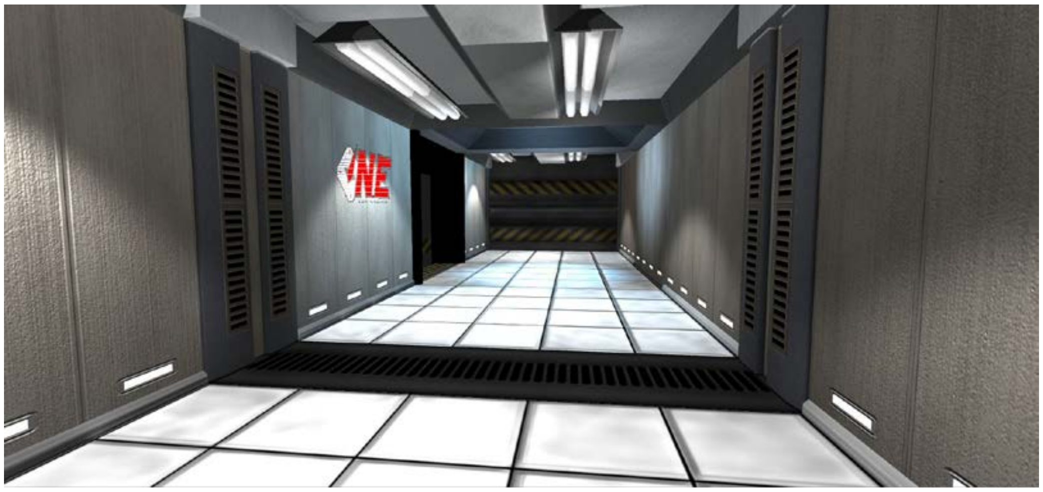


FIGURE 13: THE HALLWAYS OF THE FACILITY.

4.5.2 Biodome

The second level of the game consists of the much more open and sandbox-like biodome, where the player activates four switches to open a massive door and escape. Compared to the facility, the biodome is bright, vibrant, and shows off the natural quality of art in *Lifespan* (Figure 14).

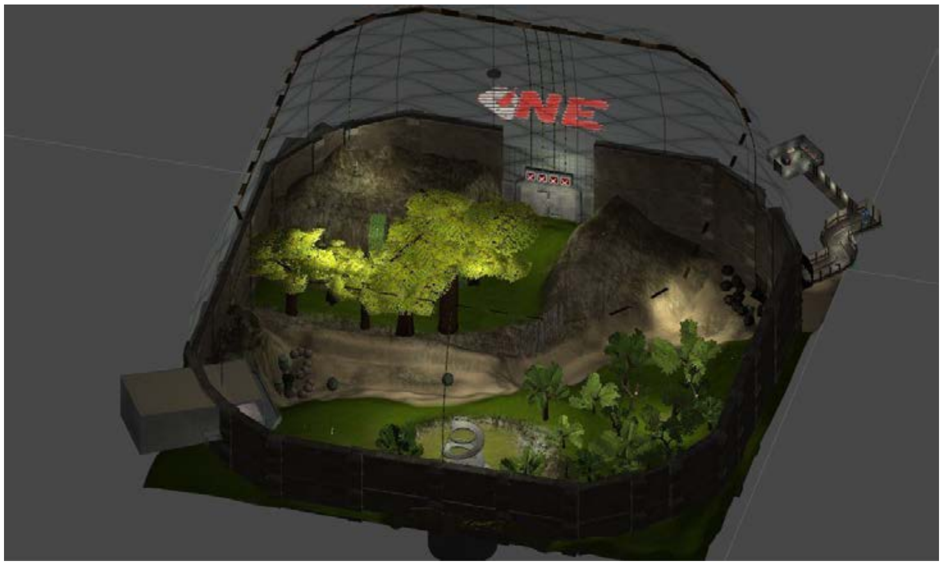


FIGURE 14: AN EXTERIOR VIEW OF THE MAIN AREAS OF THE BIODOME.

Elevator

After activating the elevator at the end of the Facility level, the player is transported into the biodome, so the the elevator is the first part of the level that the player encounters. The ride up is an opportunity for the player to get a sweeping, grand view of the entire level, as well as visual clues about how to proceed.

The biodome elevator shaft is a massive glass tube, ringed by a spiraling ramp leading to ground level. At the top of the elevator is the first of four circuit breaker switches, which open the massive biodome door at the far end of the level. A cable with red lights connects the switch to the biodome wall, and runs along the wall to the indicator lights above the door.

The elevator itself was a tricky piece to make. Being made mostly of glass – so the player could see through it into the rest of the biodome during the ascent – each piece of the elevator needed to be separated into metal and glass components. As a concave, round model, using a convex mesh collider did not fit the model, so hundreds of separate box colliders were made to fit the elevator separately.

River

Directly blocking the player's path across the biodome is a river, flowing from the Observation Room on the high ground to the low ground, where a grate blocks the Maintenance Room section from the biodome proper.

The main mass of the river was made in Maya by exporting the terrain as an .obj file and making a mesh on top of it. The part of the river at the high ground, the waterfall, was modeled separately. Both were imported into Unity as .fbx files. The water material – a standard Pro Water shader – was applied to the meshes which gave the appearance of flowing water, and a particle effect was made for the waterfall section and the rapids section towards the middle of the river to give the appearance of water foaming after hitting rocks quickly.

The waterfall section is the first section where the team used a cliff face model, put on top of the terrain for sections where the terrain looked too rounded or smooth to be realistic.

Observation Room

A room built into one side of the Biodome wall is one of the highest points of elevation in the game, and the location of the second circuit breaker switch. The Observation Room is relatively square from the outside, but ringed with a glass catwalk. By using a tree stump in front of the observation room or climbing a series of vines on the cliff faces to the right, the player can use the catwalk to enter through a window (by state reversing it into a pile of sand).

Inside, the Observation Room is in a state of disarray, much like the facility. Chairs are overturned as employees fled and work is abandoned mid-process. When the switch in pulled, every monitor in the room shows the biodome door, the player's ultimate goal, and the lights above it, which indicate how much progress the player has made in opening it.

Forest

The forest area to the right of the elevator is a lightly wooded area with trees and boulders for the player to play with or bring to other areas of the biodome to use during puzzle solving.

Jungle

In the jungle, situated across the river from the elevator, the plant life is visibly different from the ones in the forest. The trees are much taller, with a canopy of leaves only at the top, and the grass is thicker and taller.

Plateau

Slightly beyond the jungle area a plateau supports the third circuit breaker switch. By climbing some vines growing off of the plateau, the player can access the switch attached to a massive biodome relay. The relay has three curved monitors suspended from its side and a large transmission antenna extending from the center.

River Grate/Maintenance Room

Following the glowing cables running along the walls of the biodome reveals that the fourth and final switch controlling the biodome door lies beyond a rusty grate blocking the player from freely going into a river runoff area.

After breaking the grate with an accelerated boulder, the player moves through a small cave into a sewer, and from there goes into a maintenance room with the last switch in it. This area was designed to be a service and security center for the biodome (Figure 15). It is used as a proper drain and water recycle center for the river. Like the Observation Room monitors, the monitors here show the biodome door when the switch is pulled.

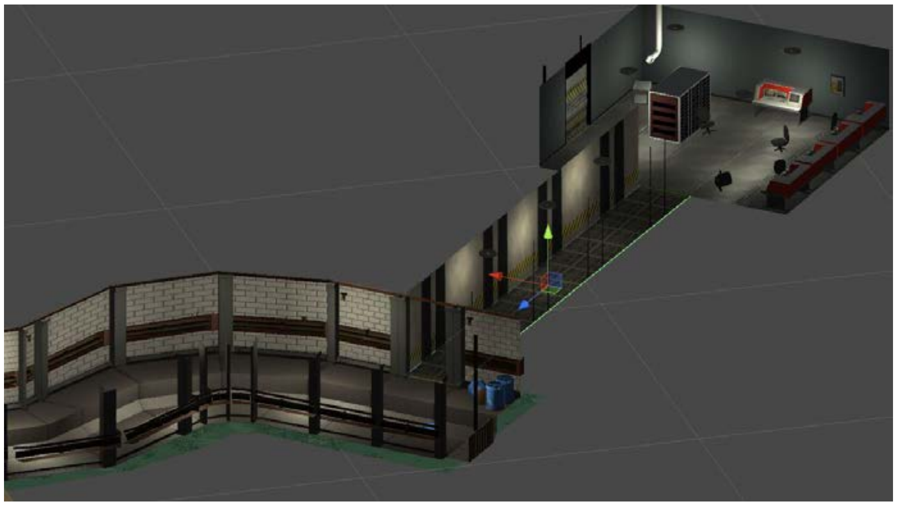


FIGURE 15: THE SEWER AND MAINTENANCE ROOM IN THE BIODOME.

Biodome Door

Preventing the player's easy escape from the N.E. Corporation is a massive locked metal door. The door is constructed using heavy steel plates held together with rivets, and locked with a spinning mechanism in the center and two large steel bolts screwing the doors into each other (Figure 16).

Above the door is a series of four lights, illuminated with big red X marks. Cables fitted with red lights run along the walls to the circuit breaker switches situated in other parts of the biodome. As the player flips these switches, the red lights on the cables and above the door turn green.

Once all four switches are flipped and the player makes their way to the door, the containment bolts unscrew, the center lock spins and unlocks, and the door itself slowly opens to reveal a huge storage room with another door that opens in a similar fashion, allowing the player to step outside into a wooded area, where the game ends.

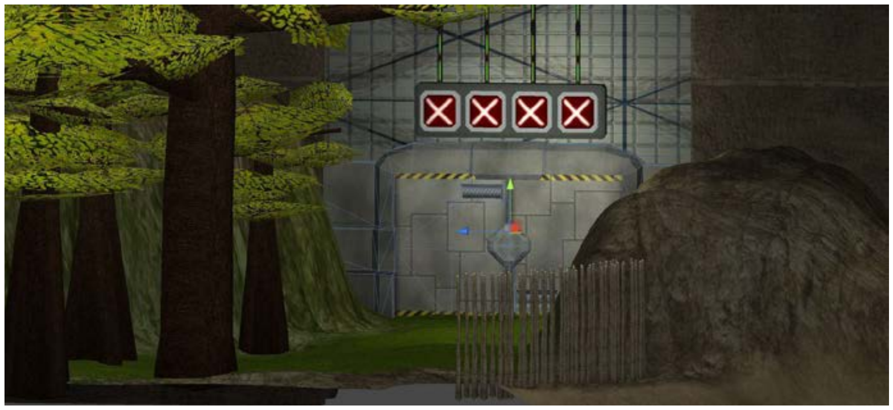


FIGURE 16: THE BIODOME DOOR PREVENTS ESCAPE UNTIL ALL FOUR CIRCUIT BREAKER SWITCHES HAVE BEEN FOUND.

4.6 GUI

The GUI for *Lifespan* was designed to resemble a graphical overlay the time manipulation device is projecting onto a visor. This display has tools for monitoring the device's powers, the current target and a console with information on the inner workings of the device. GUI elements include Power Tray Icons, Object Camera, Reticle, and Console Messages.

4.6.1 Design

The time manipulation device was built by the N.E. Corporation and uses much of their imagery in its GUI. The central theme of the design is its red glowing lines, creating the borders of each piece. Each element has dark but semi-transparent background to make anything in front of it stand out more, while simultaneously not obscuring anything directly behind it. The corporate logo is visible in many GUI elements.

4.6.2 Power Tray Icons

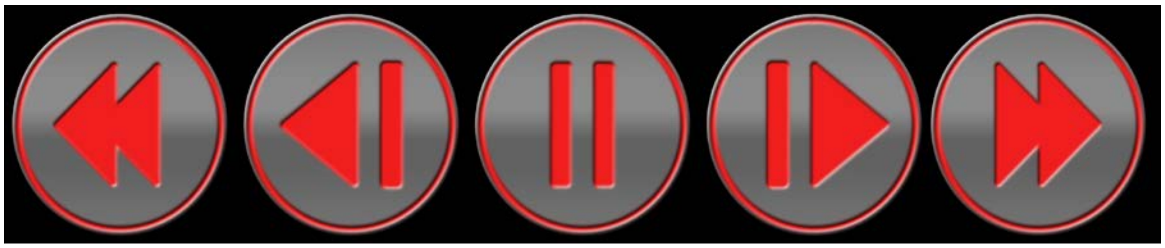


FIGURE 17: THE FIVE POWER ICONS REPRESENTING ABILITIES PLAYERS CAN USE.

Centered horizontally and aligned with the bottom of the player's screen is the Power Tray. The player looks here to know what power they have selected, as well as know what other powers they can use on any given object.

The power icons have a gray background and either a red or black highlight, depending on whether the power is selected on unselected respectively (Figure 17). Powers that the player can use on the targeted object are fully opaque on the UI, while powers that cannot be used on that object are faded and semitransparent.

Each of the five powers has a unique icon meant to symbolize what it does. Finding icons that were simple yet descriptive enough was difficult, but invaluable in assisting the player's understanding of how they can interact with the world. Some powers were easy to make icons for, such as stasis – represented by a common pause symbol. The simplest way of demonstrating the difference between reverse and forward, regardless of the distinction of state or location, was to have an arrow pointing either left for reverse or right for forward. From there, the team decided an arrow next to a line best represented a location-based power, and a double arrow best represented a state-based power.

4.6.3 Object Camera

The player has the ability to lock onto an object when targeting it in order to use a power on it while not directly looking at it. A small window in the bottom right corner shows the targeted object, along with a lock symbol when the object is locked-on or an unlocked symbol when the player's target is free. This feature helps the player differentiate between what object they are targeting or locked on to, especially in a large group of objects or from across the level.

4.6.4 Reticle

Resembling a normal crosshair when not in use, the reticle is an important way to give player feedback. When the player is targeting an object they are able to affect, the reticle expands, creating a box around the targeted object. This is the most obvious visual cue available to the player that they can use one of their powers on any given object.

4.6.5 Console Messages

Console messages in the lower right part of the screen show up when important game events happen. These messages let the player know a bit more about the game world and how the device works in-universe, but are otherwise impractical.

4.7 Challenges

An array of challenges came with creating a game in full 3D from polycount to art style, and all of them had a hand in defining the final version of Lifespan. These issues are no different from that of AAA studios; instead of letting the concerns change our concept they were embraced and worked on to create our final product. This philosophy of asset creation allowed the final product to look clean and polished without too many issues such as jagged geometry or texture blurring.

Polycount

Polycount is an issue in many games and is a huge concern for titles who are trying to go for a more realistic style and feel. An example of this in game is the arms; the original concept was to have a fully created character and then animate it from a first person perspective. Though to do this we would need to create a high polygon model that would have taken significant time and processing to have ingame. The team sat down and thought of the places where a player would be able see a reflection of the character. There were too few areas of reflection to prioritize a fully fleshed out model, thus it was scoped down to just the arm models. This also allowed for a higher poly mesh of the arms as well, giving more detail to the players perspective. Scoping decisions such as these defined the way we thought about the art in our game allowed us to create a polished and semi-realistic look to our game without straining the computers too heavily.

Creating beautiful assets with a restrictive polycount relies on a nice art style that embraces those restrictions. While creating assets, the artists needed to conform to the semi-realistic look so that the player felt like the universe was grounded in reality. Approaching this issue the team needed to use different reference images to create each asset. These reference images were used to create artistic assets grounded in reality. There were some textures that were created using pictures of actual items such as the carpet; manipulating this in Photoshop allowed for a really clean and crisp texture detail.

4.7.1 Blend Shapes

Blend Shapes had some specific rules to be followed to get the right effect. A Blend Shape had to be made of multiple models with the exact same vertex count, and the vertices needed to all be in the same order. The easiest way to deal with these issues was to make one model, then duplicate and alter it. However, once the first model is made, no changes to the geometry can be made except for moving vertices.

If vertices were too close to each other, even though they were separate vertices in Maya, Unity would optimize and merge them together, causing the Blend Shape scripts to fail due to uneven numbers of vertices. The simplest workaround that the team found was to make sure any close-together vertices were spaced far enough apart that Unity would not optimize them away. This was tested in Maya by automatically merging any vertices within a certain distance (usually 0.001 units) of each other. If no vertices were merged, the model was safe for Unity.

When creating dynamic models in Maya, the “Fill Hole” and “Bridge” tools were unsafe to use. Even when the blend shape animation worked correctly in Maya, they would often have strange visual issues while animating in Unity. The team found no fix to this issue, and instead tried to avoid using those tools in conjunction with blend shapes.

4.7.2 GUI

One concern that the team had about the GUI is its reliance on color. With red being one of the more common difficulties for colorblind people, we needed to make sure elements used light and dark more than saturated and unsaturated colors in order to be colorblind-friendly.

Design of the GUI elements was an early concern that the team had to work through. Originally elements were much squarer, which looked unprofessional. The newer GUI was designed with rounded corners in mind and looks more polished as a result.

4.7.3 Transparent Models

Due to the way transparent models are rendered in the Unity engine, a semi-transparent model won’t show up behind another model with transparency. This creates visual issues with water and water particles, which flicker when new particles are created, or two sets of glass doors across the hall from each other. A feature called sorting fudge on particle systems let the waterfall particles either always show up in front of the water mesh or always be invisible behind it, which was usually enough of a fix to keep the waterfall from looking strange.

When thick models had both transparent and opaque parts, such as the biodome elevator, the player could be positioned such that they could see through the model completely. This was unrealistic and looked bad, so every model that had both transparent and opaque parts needed to be separated into two models, and holes needed to be filled to make the glass look realistic.

4.7.4 Concave Models

Models with concave parts had issues with Unity's built-in physics, so these models needed to use convex colliders. This worked fine for subtle concave pieces that didn't really matter, such as the bumpy faces of boulders or the slits in grates, but for models that the player needed to walk through or punch objects through, like the elevator or the cave at the bottom of the river, the convex colliders caused physics issues. For these models, individual box colliders needed to be created, made invisible, and applied over the visible models to let physics work within those areas.

4.7.5 Lighting

Lighting in Unity is odd – it has different levels of importance. Importance values can be assigned to lights to change how they behave with one another. They tell the lights how to light an area based on the other lights around it. When we were lighting it was difficult to tell how the lights affected each other, so we found a setting – differed lighting – that enabled us to set every light to important, which made them all behaved the same with one another.

Lights in Unity bleed through objects. If you have an entire modular environment pieced together, even though the objects are touching and snapping together, light will go through the seams and into the environment behind it. This gave us trouble with any form of color lighting and any lights large enough to keep a room at the same luminosity without having too many light objects. Normal maps in Unity are also troublesome because they are only active when a non-baked light in shining on them. Any baking flattened the normal maps in our game and made it not look as good.

4.7.6 Grid/Snapping

In Unity there is not a way to keep grid snapping on automatically. Developers need to continuously hold down the control button to engage grid snapping. Then there was the problem that Unity has one generic grid setting and there is no way to change it in any of the settings. The tech team wrote a script to change the grid, but then it still did not matter because the snapping did not change to the newer modified grid. This meant that our placement of objects smaller than the standard grid unit was imprecise.

5. SOUND

Michael Pelissari, Nick Silvia and Collin Ogren made up the sound team of the project. The sound team was responsible for all of the auditory aspects of the game, including sound effects, from collision to static, music, and many other aspects that went into defining the soundscape of *Lifespan*.

5.1 DIRECTION

When we started the project, as a team we agreed upon that our game would excel in all categories so sound would be a big focus included in our MQP. We strived to create a real and full soundscape. We achieve said soundscape through the style of our music, sound effects, and ambience. This was all to create a sense of movement in our game. A full soundscape keeps the player involved and allows them to think the game environment is not empty and sterile. Creating crunchy and interesting sounds that resonate with the players was the design philosophy behind all of the sounds of our game.

5.2 EFFECTS

Sound effects set a scene and tie together elements in an environment. Almost every object that can be manipulated in game has some sort of collision sound or a rolling sound, while static objects rely on sounds emanating from them for them to feel like there is still motion and a liveliness to them. All of these sounds combined with the ambience of our scenes create the soundscape for *Lifespan*.

Collision

When a boulder flies against a wall a player needs to hear that impact so they can further suspend disbelief. With concepts of immersion in mind the team used the SoundManager to vary the pitch of these sounds within a predefined range. This allowed us to fill our soundscape in a smaller period of time without annoying repetition with each asset. Each of these sounds is attached to an object and move with them so they have location based volume; this allowed each object to move without the player being berated with sounds from all around the room.

Static

The static sounds allow scenes to feel more full and vibrant, so in *Lifespan* we use them to accentuate areas of the environment such as computer whirs in the offices or the sound of the water flowing in the river. There are two different forms of static sounds that we use in *Lifespan*, ambient and object based. The difference between ambient and object sounds is that ambient sounds will fill an entire area while the object sounds are attached to objects in an environment. Ambient sound in our game comes from two major sources: machines and nature. This ranges from fans running in the servers to the window rustling in the forest. We found from testing that when ambience was added to the game it added a new level of depth that kept the player engaged in the game. Some comments we received were:

- "Wow, it sounds like I am outside."
- "The facility really sounds barren."
- "That's a waterfall!"

These sounds are all combined to create full sounding scenes, such as the waterfall in the biodome.

5.3 Music

All music in *Lifespan* is original, drawing inspiration from films and computer games. The music uses real-world instrument sounds exclusively to create a soothing yet emotional and natural atmosphere for the game. Music was composed and synthesized in Sibelius before getting spliced and mixed in Audacity.

5.3.1 Composition

The music for the facility and the biodome is split into minor keys and major keys respectively. Themes for the facility emphasize solitude and apprehension while themes for the biodome play casually to relax the player. The musical themes for the biodome fall under "easy listening", encouraging the player to spend lots of time in the level. In contrast, the tenser themes in the facility level provide more emotional support to the story.

Music in *Lifespan* for the most part used strings and made frequent use of the string quartet, consisting of two violins, a viola, and a cello. This sound unifies the varying styles of music in the game and creates a very natural feeling that comes to the forefront in the biodome.

Facility

The facility uses two pieces, the test rooms piece titled "HVAC" and the halls piece titled "Alarming Desolation". "HVAC" used g and d minor while "Alarming Desolation" used f# minor. The tempo stayed the same throughout both pieces of 100 beats per minute (bpm) with long drawn out bass lines. "HVAC" uses pan pipes to emulate an HVAC system, providing ambience and an empty feeling that matches the empty facility. A clarinet plays a somber melody on top of this to emphasize the loneliness of the place. "Alarming Desolation" tweaks the theme a bit by adding tension through the use of oscillating flute notes in an alarm-like fashion and a triangle that increases tempo before cutting out. A violin plays two squeaky notes in the second bar of a two bar phrase to startle the player, adding anxiety to the atmosphere. This piece continues playing with different elements that mimic alarms until it ends in a climax at the end where the triangle chimes ridiculously fast and a timpani pounds twice in succession. This finale originally signalled that the player was dead at the end of the ten minute time limit we had set, but even without the timer, it serves as a great tension builder that fails to resolve, leaving the player feeling very anxious.

Biodome

The biodome uses several pieces. "The Biodome" opens flourish when the player enters on the elevator. After the piece finishes, "Plateaus and Plains" begins. "Cheery Avians" plays in the forest, "The River" plays along the river, and "Jungle Fever" plays in the jungle area with the tall grass and the tall trees. When the player walks out of the biodome into the outside, "Winning Freedom" plays to close the game with a congratulatory exaltation.

When the player enters the biodome from the dark, underground, and empty facility, he or she is greeted with a beautiful scene of pristine artificial wilderness. "The Biodome" is supposed to support that breath of fresh air that the player should feel when entering this level. The G major piece starts out low with a bass line and then begins a drum roll on the timpani to increase anticipation. The player is rewarded with a splash of a cymbal as the player's view peeks above the ground. The stings rise with the player to not only follow the motion of the elevator, but also support the player's rising spirits. This ends in a regal climax accompanied by a trumpet to highlight the grandeur of the environment. The emotion is elating and refreshing. This piece plays at 100 bpm.

After “The Biodome” completes, “Plateaus and Plains” plays Allegro in D major. The melody of the string quartet is written in pentatonic scale, giving the themes a feeling of traversing the highlands with wide open space, much like the plateau and sunflower field where the piece plays. Inspiration for some of these themes comes from the soundtrack for *The Lord of the Rings*.

In front of the player, a river flows across the biodome. Upon approaching this stream, a gentle string quartet plays long chords as a flute accompaniment plays a soft melody. The melody changes between a fluttering, bubbling brook and a meandering stream. The fluttering melody is inspired by the opening of “Die Moldau” by Smetana. The D major and A major piece is calm and relaxed at 80 bpm like the water in the river.

To the right of the player as he leaves the elevator, he can see a grove of trees that form a man-made forest. The music that plays here also contains long chords from a string quartet, but the accompaniment comes from a duo of flutes instead of the solo in the river piece. Together, they play alternating notes to mimic the songs of birds that one finds in a forest. This piece in D and G major is also calm, but the flutes are faster at some points to liven the forest. The piece therefore plays at an Andante pace with no specific metronome mark given.

The other area in the biodome with a concentration of trees is across the river, called the jungle. To give a jungle-like atmosphere, the music uses only percussion instruments and a recorder. Percussion instruments included a whistle, castanets, two guiros and a set of four congas. The percussion provided background rhythms while the recorder played long notes on top in C major and G major. While the recorder plays in these keys, the piece is largely atonal and is meant to have little formal structure other than a steady rhythm. Most of the piece is in 4-4 like the other pieces, but one part is in 3-4. In addition, rainstick recordings were mixed in during post-production in some parts of the piece to emulate the rustling of leaves in the wind and the trickling of rain. The idea behind these sounds it to create an exotic atmosphere reminiscent of a jungle.

The final piece of the biodome comes at the very end and is technically not in the biodome at all. This is the endgame piece of four bars. A string quartet plays rising G major chords with Violin I playing four rising quarter notes in the first bar. These four notes introduce the trumpet and timpani for the next three bars. The trumpet doubles Violin I, and the timpani beats on the down in the second and third bar and twice in succession at the end of the fourth bar. This creates an escalation followed by an exhalation, much like a triumphant sigh of relief to illustrate the breath of fresh air that comes with escaping and finishing the game.

5.3.2 Design

Music in *Lifespan* is dynamic. Each piece described above is a set of music tracks that can play continuously into the other tracks in the set. When the player enters a trigger, the attached trigger script instructs the Sound Manager to look up the associated tracks and randomly pick one. When that track nears the end, the Sound Manager randomly picks another track from a list that is tied to the currently playing track. In this way, we can control what tracks can follow the current track, enabling some control over the flow of the music while keeping it dynamic.

One exception to the track set is in the hallways of the facility. In this one location, one music track plays continuously for ten minutes. This reflected an initial design decision that the play would die after spending ten minutes in the area if he did not move on. This decision was reversed however, so the track now repeats itself once it finishes.

The data for the music resides in XML files that the Scene Manager loads into the game with the scene. Each scene event contains a sound sub-event that holds the list of tracks. Each track holds its own list of tracks that specify which tracks can follow it. The tracks also contain data for fade-in and fade-out times, volume, and pitch.

While the XML contains track data, the triggers in the scene control silence. Silence, specifically the absence of music, occurs in several places in the game. When the game first starts, the player is in a server room where the only sound comes from ambience. Music does not begin playing until the player encounters the first trigger when he obtains the device. Silence also occurs in the biodome when the player enters less natural areas like the observation room, the maintenance tunnel, and the loading dock. This silence occurs because the player enters a trigger with no tracks specified, but the trigger does call an event which fades out all music in the level. Music starts playing again when he enters a trigger that specifies a set of music tracks.

Music triggers do more than simply specify the track to play. In the facility, the trigger for the hallway music disables the music for the test chambers. This reflects the irreversible activation of the alarms in the facility and allows the whole piece to play through. In the biodome, triggers precisely control the opening sequence by crossfading from one track to the next as the elevator raises the player upwards. These triggers act as cues for the sequence, much like music for a film is cued in each scene. These cues were necessary over simply timing one whole track because the variation in frame rate can change the speed of the rising elevator. The triggers therefore rely on player position rather than time.

5.4 Challenges

5.4.1 Sound Effects

Creating the sound effects was a relatively simplistic process due to having access to a full sound library. The main issue was finding sounds that could fit our needs by going through and listening to each one in the library. Listening to each sound became extremely repetitive but allowed us to find sounds that could be used for something completely different, such as an ambient track that we could speed up to make a device noise. In the beginning, each sound was created by either using Foley or synthesizing them in programs. This whole process of sound creation was a large time sink and luckily access to the sound library alleviated the burden later on in the project.

Other issues we faced were in balancing the volume of each sound but due to the SoundManager this process was exceedingly easy and took little time. Within the Unity engine there is a unique limitation with putting in ambient environment or music sounds. Sound objects in Unity propagate from the direct center, as if coming from a radio. This causes problems with trying to place ambient sounds in the facility and biodome so the same level of volume can be heard everywhere or conformed to square halls and rooms. The challenges of sound ended up being solved mostly with creativity and cunning use of what we had at our disposal.

5.4.2 Music

The major challenge with music involved making the music flow. As each piece consisted of randomly selected tracks, each track had to transition harmonically and melodically into the other tracks in the set. Controlling which tracks could follow a track helped make this easier so that a track did not have to transition well into *every* other track. Tracks also tended to either start and end on the tonic chord of the scale, which happens with the river music, or a track would end on a dominant chord and the next track would start on the tonic chord. These are common chord progressions that sound very satisfying to the human ear, which is why they were used in transitions. To make transitions even smoother, the music pieces make frequent use of strings for background chords. Another element that helped music transition, especially between sets of tracks was crossfading them in-game. This created smoother transitions between different styles of music.

6. Project Management

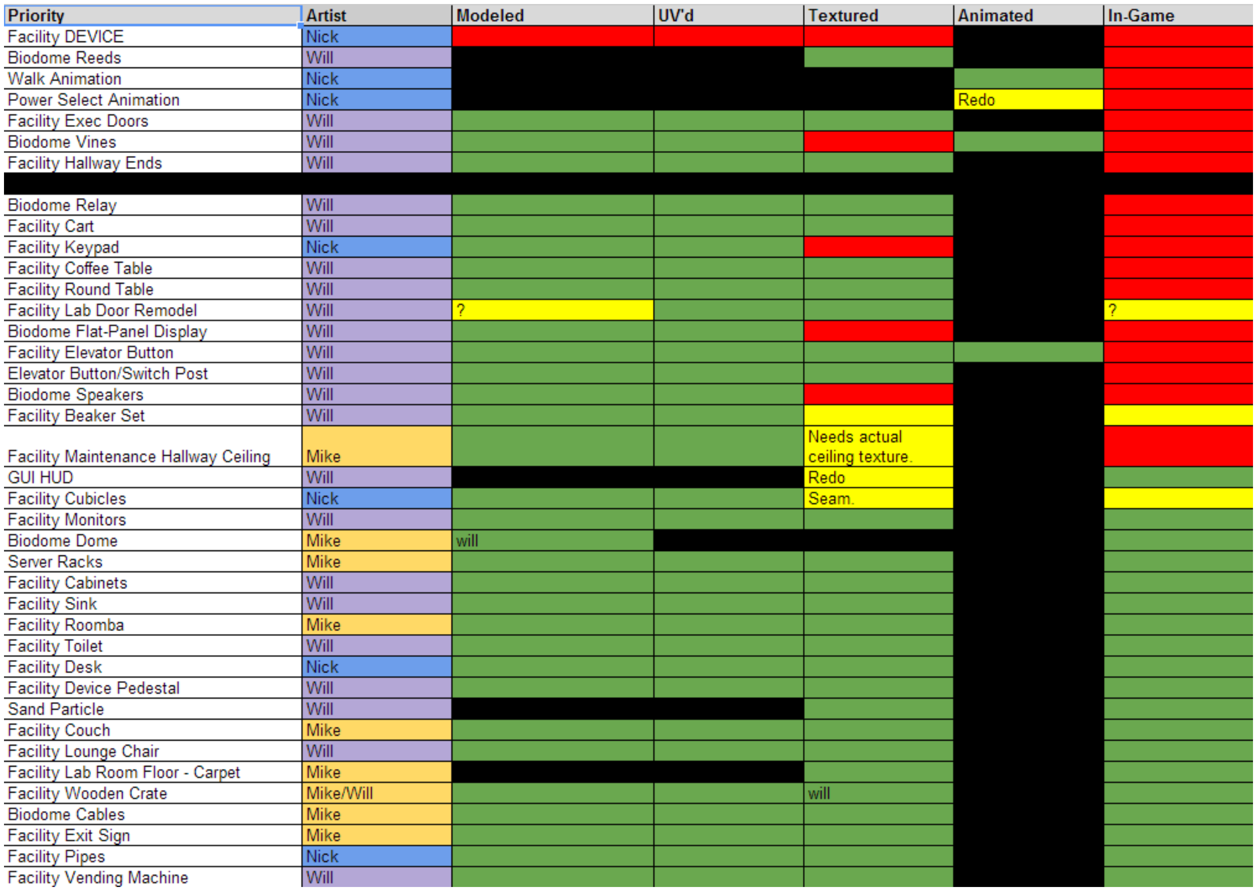
Michael Grossfeld and Matt Tomson were the producers for the development of *Lifespan*. They assisted team members by providing instruction, managing content, and other administrative tasks.

6.1 Scope

After the initial mechanic design, we came up with many locations and environments that the game would be interesting to play in. We came up with roughly five levels, mostly taking place in the jungles of Brazil. Making all of these levels would not be feasible in the time-frame available, so the game needed to be scoped down. This led to two levels, one inside and one outside. We then needed to populate these levels, so we came up with lists of assets that we needed and wanted. We started creation of the game in the fall and quickly realized that there was too many assets we wanted to create and that we only had enough time and people to do some of them. We then discussed prioritizing the assets at the start of B Term to better allow us to scope our game. We were able to cut several sections of the biodome from the bottom of the list, knowing we did not have the time to complete them and that they were the least interesting and important sections of the level. The facility also underwent scoping, as many of the assets we had planned to create ended up cut due to time constraints.

6.2 Planning

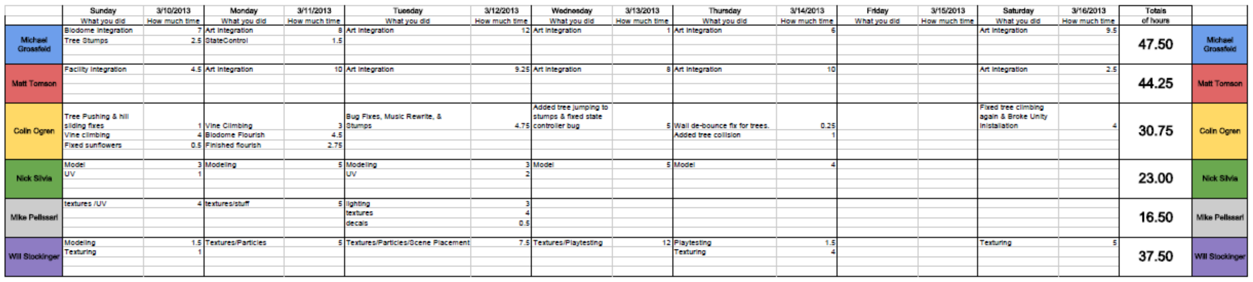
Despite having an above average size team, we were able to complete the game we set out to make with minimal complications through a set of project management tools that allowed the producers to plan out the game on a weekly basis. One of the tools that we utilized was a priority list for art assets kept in Google Docs (Figure 18). These lists contained all of the assets that we could want to put in the game. Created in B Term, we were able to add and remove items from this list. Items at the bottom were scoped out as time passed and sections needed to be cut. During development, new assets came to mind and were added to the list in an appropriate position. Artists were able to check what assets needed to be created next without having to check with a producer or the lead artist. In order to make sure only one artist was working on a specific asset at a time, a column in which an artist could claim an asset was added to the spreadsheet. These priority lists proved invaluable in the later portions of development where assets needed to be cut due to time constraints.



Priority	Artist	Modeled	UV'd	Textured	Animated	In-Game
Facility DEVICE	Nick	🟥	🟥	🟥	⬛	🟥
Biodome Reeds	Will	⬛	⬛	🟩	⬛	🟥
Walk Animation	Nick	⬛	⬛	⬛	🟩	🟥
Power Select Animation	Nick	⬛	⬛	⬛	Redo	🟥
Facility Exec Doors	Will	🟩	🟩	🟩	⬛	🟥
Biodome Vines	Will	🟩	🟩	🟥	🟩	🟥
Facility Hallway Ends	Will	🟩	🟩	🟩	⬛	🟥
⬛	⬛	⬛	⬛	⬛	⬛	⬛
Biodome Relay	Will	🟩	🟩	🟩	⬛	🟥
Facility Cart	Will	🟩	🟩	🟩	⬛	🟥
Facility Keypad	Nick	🟩	🟩	🟥	⬛	🟥
Facility Coffee Table	Will	🟩	🟩	🟩	⬛	🟥
Facility Round Table	Will	🟩	🟩	🟩	⬛	🟥
Facility Lab Door Remodel	Will	?	🟩	🟩	⬛	?
Biodome Flat-Panel Display	Will	🟩	🟩	🟥	⬛	🟥
Facility Elevator Button	Will	🟩	🟩	🟩	🟩	🟥
Elevator Button/Switch Post	Will	🟩	🟩	🟩	⬛	🟥
Biodome Speakers	Will	🟩	🟩	🟥	⬛	🟥
Facility Beaker Set	Will	🟩	🟩	🟨	⬛	🟨
Facility Maintenance Hallway Ceiling	Mike	🟩	🟩	Needs actual ceiling texture.	⬛	🟥
GUI HUD	Will	⬛	⬛	Redo	⬛	🟩
Facility Cubicles	Nick	🟩	🟩	Seam.	⬛	🟨
Facility Monitors	Will	🟩	🟩	🟩	⬛	🟩
Biodome Dome	Mike	will	⬛	⬛	⬛	🟩
Server Racks	Mike	🟩	🟩	🟩	⬛	🟩
Facility Cabinets	Will	🟩	🟩	🟩	⬛	🟩
Facility Sink	Will	🟩	🟩	🟩	⬛	🟩
Facility Roomba	Mike	🟩	🟩	🟩	⬛	🟩
Facility Toilet	Will	🟩	🟩	🟩	⬛	🟩
Facility Desk	Nick	🟩	🟩	🟩	⬛	🟩
Facility Device Pedestal	Will	🟩	🟩	🟩	⬛	🟩
Sand Particle	Will	⬛	⬛	🟩	⬛	🟩
Facility Couch	Mike	🟩	🟩	🟩	⬛	🟩
Facility Lounge Chair	Will	🟩	🟩	🟩	⬛	🟩
Facility Lab Room Floor - Carpet	Mike	⬛	⬛	🟩	⬛	🟩
Facility Wooden Crate	Mike/Will	🟩	🟩	will	⬛	🟩
Biodome Cables	Mike	🟩	🟩	🟩	⬛	🟩
Facility Exit Sign	Mike	🟩	🟩	🟩	⬛	🟩
Facility Pipes	Nick	🟩	🟩	🟩	⬛	🟩
Facility Vending Machine	Will	🟩	🟩	🟩	⬛	🟩

FIGURE 18: THE ART PRIORITY LIST.

Another tool used for managing the team was weekly timesheets (Figure 19). After any member of the group worked on something related to the implementation or the design of the game, they would log their hours into the timesheet. These timesheets also kept track of the tasks that each team member had spent time on. The producers could then check to see who was working and what was being worked on, and then adjust for the next week. For example, if one person was working much less than the rest of the team, the producers could inform said team member and get them to put in more hours in the following week. If a member spent many hours working only on one asset, it could be that the asset was far too complicated than originally planned, and had to be adjusted in the priority list.



	Sunday	3/10/2013	Monday	3/11/2013	Tuesday	3/12/2013	Wednesday	3/13/2013	Thursday	3/14/2013	Friday	3/15/2013	Saturday	3/16/2013	Totals	
	What you did	How much time	What you did	How much time	What you did	How much time	What you did	How much time	What you did	How much time	What you did	How much time	What you did	How much time	of hours	
Michael Grossfeld	Biodome Integration Tree Stumps	7 2.5	Art Integration StateControl	8 1.5	Art Integration	12	Art Integration	1	Art Integration	6			Art Integration	9.5	47.50	Michael Grossfeld
Matt Tomson	Facility Integration	4.5	Art Integration	10	Art Integration	9.25	Art Integration	8	Art Integration	10			Art Integration	2.5	44.25	Matt Tomson
Colin Ogren	Tree Pushing & hill sliding fixes Vine climbing Fixed sunflowers	1 4 0.5	Vine Climbing Biodome Flourish Finished flourish	3 4.5 2.75	Bug Fixes, Music Rewrite, & Stumps	4.75	Added tree jumping to stumps & fixed state controller bug	5	Wall de-bounce fix for trees. Added tree collision	0.25 1			Fixed tree climbing again & Broke Unity installation	4	30.75	Colin Ogren
Nick Silvia	Model UV	3 1	Modeling	5	Modeling UV	3 2	Model	5	Model	4					23.00	Nick Silvia
Mike Pellssari	textures /UV	4	textures/stuff	5	lighting textures decals	3 4 0.5									16.50	Mike Pellssari
Will Stockinger	Modeling Texturing	1.5 1	Textures/Particles	5	Textures/Particles/Scene Placement	7.5	Textures/Playtesting	12	Playtesting Texturing	1.5 4			Texturing	5	37.50	Will Stockinger

FIGURE 19: EXAMPLE TIMESHEET.

Group work sessions were another important part of completing *Lifespan*. The whole team meeting in the same room to work had many benefits. First, it allowed for better communication. The whole team was able to discuss any issues that arose as well as make design choices. Work sessions also led to better feedback. Team members could get opinions on the assets that they were creating while they were creating them. Productivity was also increased during the work session – as a team, we worked much more efficiently and effectively together in the work sessions than separately.

6.3 Challenges

During the course of development, the producers faced many challenges. Most of the issues were found with enough time to either correct the mistake or change the production strategy to work better. There were also some challenges that although noted, were not able to be solved in time.

6.3.1 Early Time Management

Time management of the team was probably the most difficult challenge, as wrangling six people into getting work done is never an easy task, especially when each person has a number of courses they need to pass each term in order to graduate. Initially, we assigned people to tasks, gave them a soft due date, and had them work on the task on their own. Very quickly we realized this was not the best practice, as most of our work was being done only when the entire team met together. Starting towards the end of A Term we began three weekly work session – two mandatory, one voluntary. This enabled us to get the most out of our team, as we produced more work when working with one another.

6.3.2 Priority Lists

Additionally, task structure was a difficulty we faced. In the beginning, tasks were routed through the producers arbitrarily and then assigned to whomever would produce the work efficiently. Unfortunately this meant that tasks, such as creating specific assets or coding certain scripts, were not stored in some prioritized list. When someone finished a task, they didn't often know what came next until a producer told them, which meant that there were periods of time that no content was being produced.

Starting towards the end of B Term, prioritized lists were put up on the Google Drive folder. These lists were color coded to let the entire team know which assets how far along the pipeline each asset was. This helped figure out exactly what assets were lagging behind and to also show what was the cause of their delay. There were times where an asset had been created but the art integration team had not been informed – this meant that the asset was just sitting in the repository, waiting to be used.

6.3.3 Timesheets

Aside from priority lists, we also used timesheets to monitor and track the team's progress. These timesheets were by user input; if a team member forgot to insert their hours, they would often not end up on the timesheet. While the timesheets were meant as a way for our advisors and the producers to keep track of weekly work, it ended up only being used as a metric for competition. More of us valued the timesheets as a quick study of who was working the most on *Lifespan* in a given week, and then attempt to put in more hours than that person. The timesheets ended up being mostly useless and a burden for the team to remember to fill out, which was time that could have been spent on other things.

6.3.4 Communication

Communication also proved to be troublesome. Our main form of communication was via email, which was not being checked constantly by all members, nor replied to immediately. Gaps in knowledge quickly formed as some people had read emails and others hadn't. These gaps were rectified during work sessions, however. Other forms of communication included text messaging, which was faster than emails but more difficult to execute due to there being six different members to communicate with individually and no group option available. The best form of communication was face-to-face discussion in our work sessions. These allowed us to deliver quick feedback and information to the entire team without having to wait on anybody in particular. Face-to-face discussion was our preferred method of communication but unfortunately not always available.

6.3.5 Scope

Another challenge we faced that wasn't team related was our scoping of the project. Initially in our design the facility consisted multiple floors and the biodome was a number of modular outdoor pieces that would be randomly arranged for the player to escape from. This was quickly scoped down to be just the facility with two levels and a biodome level. We continued to scope downwards, but that initial, larger scope proved to be difficult to shake for quite a while. This led to us developing some systems and assets we didn't end up using which was a case of poor project management on the part of the producers.

7. PLAYTESTING

One crucial phase of this MQP was playtesting sessions. About a dozen people playtested the game, giving feedback, finding issues, and proposing changes. The playtesters came from varied backgrounds, but most were students of Worcester Polytechnic Institute. There were men, women, gamers, and non-gamers. These playtests were run during the early beta phase of development, after the biodome had been roughly implemented in C term. This gave the players a full game to play, with the exception of the final room that the game now contains.

7.1 PROCEDURE

Playtests were set up to be run by one group member and one tester. The group member read an introduction to the tester about *Lifespan* and the ideas behind it. The tester would then start playing, talking aloud during gameplay with thoughts, comments, concerns, and suggestions. The group member would write down these notes in a notebook. After completing the game, the tester filled out a survey of questions pertaining to the gameplay experience.

7.2 FACILITY

One issue that was found during playtesting is that players had trouble understanding what the powers were doing when they first encountered them in the test chambers of our facility, which served as a tutorial section. This was remedied by adding text explanations printed out by the device on the user interface. These text explanations tried to give players a short yet informative passage explaining the power and how to use it.

An issue encountered during the first playtests was that players were unsure how to navigate the facility. Players would run around in circles without realizing that they were in the same places they had been before. The walls, floors, and doors are the same material throughout the facility, so there were no distinguishing characteristics. Adding exit signs and distinguishing clutter to some of the doors, halls and ceilings led to players following the signs to reach the exit. This allowed players to more easily find the exit to the level, but also made them less likely to explore the rooms that were contained in the level. Seeing as the biodome was the more important level in the team's opinion, we decided that this was a tradeoff we were willing to make.

7.3 BIODOME

Navigating the biodome to find the four switches proved to be a issue with the game as well. Players could find the first switch, but had some trouble locating the other three. The team fixed this by adding lit cables that the player could follow, stretching from the door at the end to switches. These cables were lit red for switches that still needed to be activated or green for already activated switches. At any time the player could look at the walls of the biodome to determine where they've been or where they need to go next.

Some players confused the first switch for an elevator switch (to raise/lower the elevator) but when we added a different model that the players used to activate the elevator, this confusion was no longer an issue.

7.4 Powers

Some more issues encountered during playtesting concerned the functionality of two location-based powers: forward and freeze.

Players were confused as to why the location forward power did nothing on a object that wasn't already moving. The reason for this previously was that location forward amplified an object's current trajectory, so multiplying a velocity of zero by anything would still lead to zero. This was not made clear to the players and they expected differently. Therefore, the mechanic was changed so that when an object is not moving, it is given a velocity in the direction that the player is facing. When an object is already moving, location forward behaves the same as before, moving it faster in the same direction.

Location Freeze was the other power that had issues with its design. Before playtesting, players would freeze an object, then activate one of the other location powers on it. This caused the object to become unfrozen. The other two location powers did not act on the object. This was not a design decision made by the team, but rather an oversight by the team which was never handled until it was found during playtesting. Players expected that reversing an object while it was frozen would move it back. With the redesigned location forward, players expected the object to move forward when location forward was activated on the object when it was frozen. We then changed it to work as players expected, such that activating a location forward or reverse on a frozen object will both unfreeze it and activate the selected power on it.

7.5 Results

Opinions of the game were generally positive. Players enjoyed playing the game and thought that the mechanics and puzzles were interesting and engaging.

Making playtesting a high priority for us proved to be beneficial to the design and development of *Lifespan*. The project would not be where it is without the playtests that we conducted. We have a deep appreciation for the playtesters who took the time to run through our game – anywhere between twenty and fifty minutes for a testing session. *Lifespan* would not be nearly as polished if it weren't for those hours spent by our playtesting volunteers.

8. Post-Mortem

Looking back at *Lifespan*, there are numerous things we did well, we could have done better, and we would definitely do differently next time.

8.1 What Went Well

8.1.1 Team Dynamics

There were a number of things that went quite well for us during the development process for *Lifespan*. Chief among them was our team dynamic – despite being a six-person team, we all knew and worked well with one another. This allowed us to get work done more quickly and efficiently than it would have been had we not worked so well together. Additionally, each individual person's strengths were known and recognized which allowed for task assignment to go smoothly without any wasted time.

8.1.2 Unity

Unity worked well for us as a team. Most of us had familiarity with Unity from IMGD 4000/4500, which meant that we could jump right into work as soon as the MQP started. The Maya-like interface and forgiving asset integration workflow was friendly for the artists and easy to use for the techs, as opposed to other engines such as UDK. Only one member of the team wasn't familiar with Unity before the start of the project, and he spent some time researching it and working on tutorials in order to come to the rest of the team's level of understanding.

Unity also has a good amount of documentation with their API's, which meant that features previously unused could quickly be researched in full and then implemented. Features such as AudioClips get and set data functions were invaluable to the music playback in the SoundManager and were previously unknown features to the tech team.

8.1.3 Planning

Our early start on planning, in our Junior year, helped us get a decent amount of the work out the way. Design work, such as fleshing out the environment, characters, and the world of *Lifespan*, was done in D Term of 2012 and allowed us to get started on actual development in the following A Term. This meant that we had an extra seven weeks of development time that was spent getting systems up and running before the majority of our assets came down the pipeline.

It also helped that we were able to get our initial systems up quickly. The major hurdle for *Lifespan* was the blend shape controller in Unity, which allowed for the real-time morphing of our objects. Without this system, half of our mechanics would not have worked, so it was very beneficial for this system to have been functional before A Term.

8.1.4 Playtesting

During the actual development process, it proved useful for us to have continual playtesting. These playtests reaffirmed numerous concepts of ours, such as the "fun" rating of our game and how well the players understood the various abilities. The playtests also showed us the flaws – certain aspects of the mechanics needed to be tweaked to follow better with the flow of the game and with the players' expectations. These repeated playtests allowed us to iteratively change our game for our target players and in doing so make a better game.

8.1.5 Scope

We also continued to adjust our scope as we developed. As mentioned earlier, our initial designs for *Lifespan* included more floors to the facility and a variety of outdoor areas. These were cut early on, and more sections and levels were redesigned and adjusted as time went on so that we could produce a quality product. This process of design, develop, scope, and then iterate left us with most of our original game cut or changed, but left us with a focused product.

8.2 What Could Have Gone Better

8.2.1 Unity

Despite the number of things that went well, there were also things that could have gone better. For instance, while Unity was familiar to us, Unity did not prove to be the best choice with everything. Lighting in Unity isn't as nice as the lighting in another engine, such as UDK. Lights that are placed too close together in Unity games select only one to render by turning off the other lights nearby. For small areas in our facility, it meant that there were a number of lights that provided marginal help but still helped with performance issues. Additionally, because of the modular development of our levels, the pieces didn't always connect perfectly. Lights bled through those seams which caused incorrect color flooding in some of the rooms in the facility.

Prefabs, which are prefabricated objects in Unity, also did not work how we expected them to. Our belief was that prefabs worked hierarchically so that prefabs in other prefabs would update properly. This was an incorrect assumption of ours and it cost us time when we realized our misunderstanding. A prefab in a prefab is assimilated by the parent and loses its connection to the original. The new prefab loses the ability to auto-update, which meant that when each room in the facility was made into a prefab, all of the building pieces and objects lost their connections to the original prefabs and instead became separated instances of that prefab. When we made a change to those prefabs we would have to go to each room and swap out the older objects for the newer objects. Had we known how prefabs actually worked, we would have structured our project better.

8.2.2 Unused Assets

In the end, we accumulated scores of assets ranging from models and textures to sounds, but not every one of them ended up in game. These assets were created before their purpose was cut due to scope. Any leftover assets became wasted time which could have been spent on more useful things. Had we planned better and scoped things out earlier, these assets would not have been created, and the time would have been allocated to something else.

8.2.3 Art Integration

The assets we did create were more difficult to get into game than we assumed. There ended up being a number of both art and technical steps that were necessary to integrate a single art asset, and our communication regarding the integration process wasn't as good as it could have been. Generally, an art asset was created and placed into the repository for integration. The next time the entire team met, the new art asset was discussed and then a member of the tech team would spend the time to integrate it. If there were any problems with the integration, the tech would relate it to the art team, who would then go to fix it. This process would repeat until the final version of the asset was in-game. This pipeline often caused a number of delays and became extremely inefficient with specific assets, such as blend shapes.

Blend shapes required steps within Maya, such as binding the model as a skin to a joint, that were generally forgotten and then had to be done by the tech team, often after many other steps had been already completed. Backtracking would then have to be done, which spent more time than necessary. Then once the blend shape was in Unity there might be some tweaks or fixes that would have to be done and the process would repeat again.

8.2.4 Playtesting

Playtesting was an integral portion of *Lifespan.* It allowed us to iteratively design our game to be more fun and interesting to the players. One thing that we would have done differently regarding playtesting was getting more playtesters and starting earlier. We only tested our full game, when we could have just as easily tested parts of the game with playtesters. This would have solved both issues that arose from playtesting. Since our game was long, slightly under an hour for the average playthrough, some testers were unable to devote the time necessary to complete whole playtests. Testing only the biodome or only the facility would cut playtime down to allow users to complete the test quicker. Another part of playtesting our game was that the full build was not up until late in development. The facility was done much earlier in the year, which should have been playtested separately of the biodome. We were only able to get a few playtests of the facility before the biodome. More testing should have been done while waiting for the implementation of the biodome to reach a playable state.

8.3 What We Would Do Differently Next Time

8.3.1 Unity

Learning from the things that could have gone better, there were a number of things we would have changed or done differently if we were to go back in time and do this project again. With Unity, we would have spent more of our free time over D Term 2012 and the summer to research the many benefits and faults within the Unity API. Knowing how coroutines actually worked would have allowed us to avoid using them for sound capacity early on instead of requiring us to refactor the code to use threads. Early on, we would have stressed importance values with Unity lighting had we known of their existence and value. Additionally, the difference between streamed and memory-loaded audio files would have been useful. Even though you can select to load an audio file as memory-loaded, if it is any file-type other than a WAV, it is streamed in by Unity and therefore cannot be directly manipulated by the AudioClip class.

Knowing these different things about Unity might have even led us to use a different engine altogether. The idea of using UDK was floated around for a while, but our relative unfamiliarity with it was the decision not to use it. Another engine we inspected was HeroEngine, but for the same reason as UDK we decided to forgo using it. Despite our grievances with Unity, we would have still chosen to use the engine regardless. However, we would have taken the time to upgrade Unity from 3.5, which is the version we started the project with, to 4.0, which has a number of features. One such feature is the ability to bake normal maps into lightmaps, which was a reason we never used baked lights and was one of our biggest problems with Unity lighting.

8.3.2 Blend Shapes

With the extra research we would have also tried to find an easier and better implementation for Blend Shapes. The number of steps in getting blend shapes to work were tiring and complicated when there wasn't any need for it to be. The first version of controlling blend shapes from Maya to Unity was the one we stuck with because it was the only implementation we found that we could work with. More research would have been spent finding alternative methods of moving vertices in Unity and getting the vertex movement data from Maya into a format that we could utilize easily.

8.3.3 Scope

As one of our largest problems, we would take a closer look into the scope of our project from the start and effectively scope down much earlier on in the process. While our lofty goals drove us to deliver a final product that exceeded all of our expectations and the expectations of the public, we could have further increased our product quality with better scoping. We should have looked to get the minimum vital product completed first, and then expanded from there. That would have allowed us to create smaller system and then iterate on them, increasing our productivity and ensuring that we would have less unused assets in the end.

Appendix A: Concept Art

FIGURE 20: CONCEPT ART FOR THE MECHANICS OF LIFESPAN.

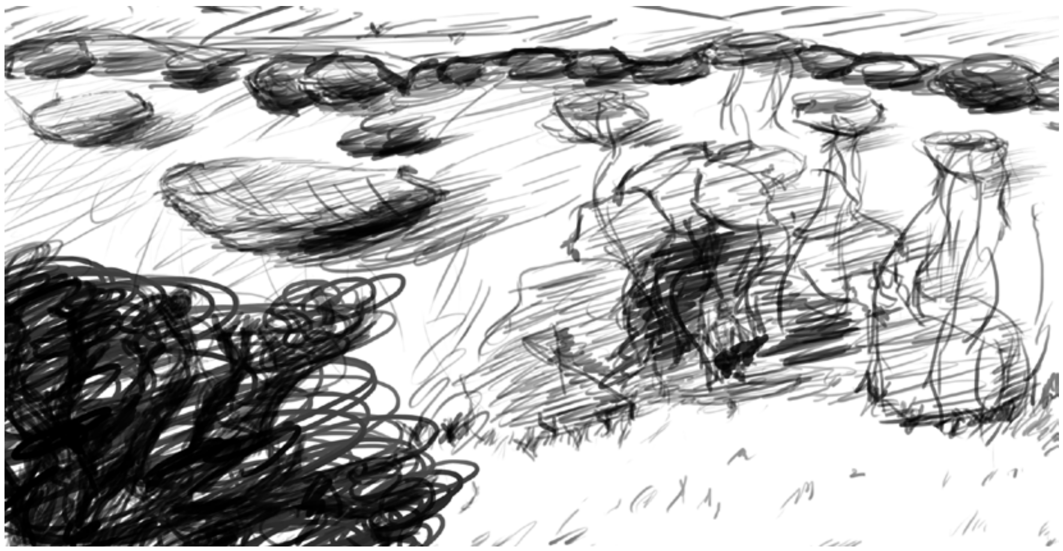


FIGURE 21: CONCEPT ART FOR THE SCRAPPED CAVE AREA IN THE BIODOME.

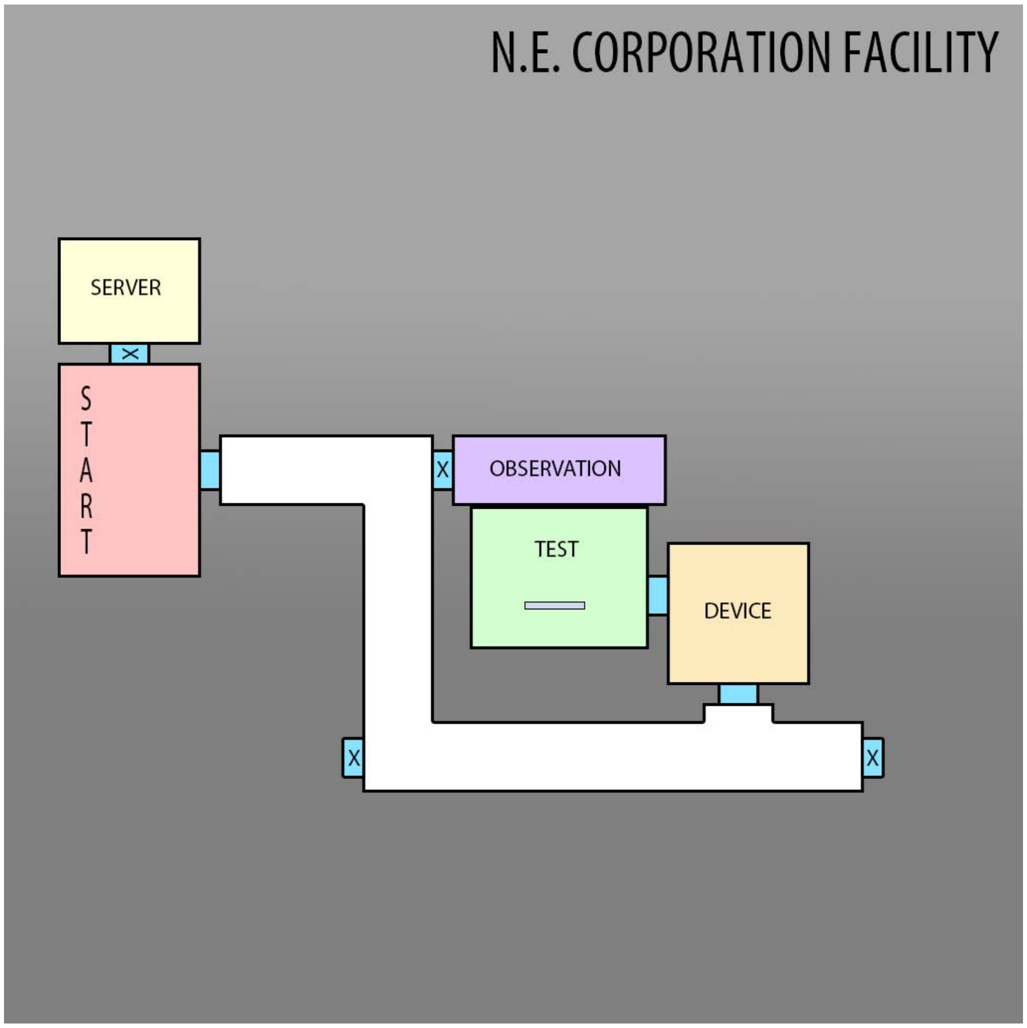


FIGURE 22: FIRST CONCEPT FOR THE FIRST FLOOR OF THE FACILITY.

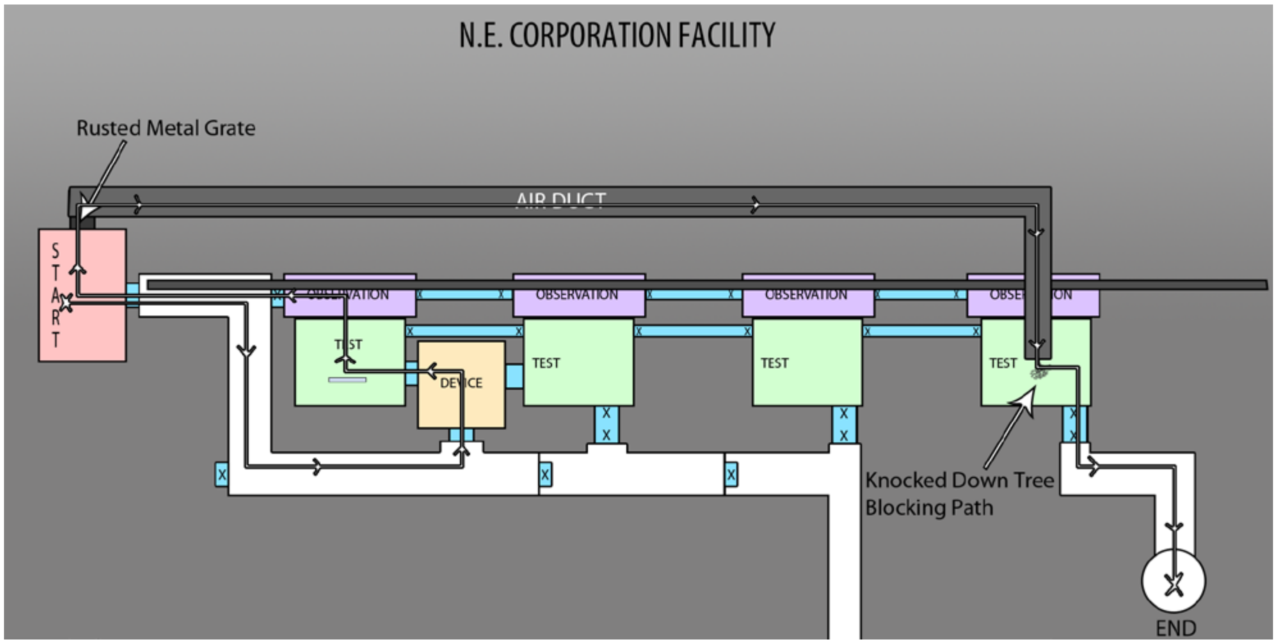


FIGURE 23: SECOND CONCEPT FOR THE FIRST FLOOR OF THE FACILITY.

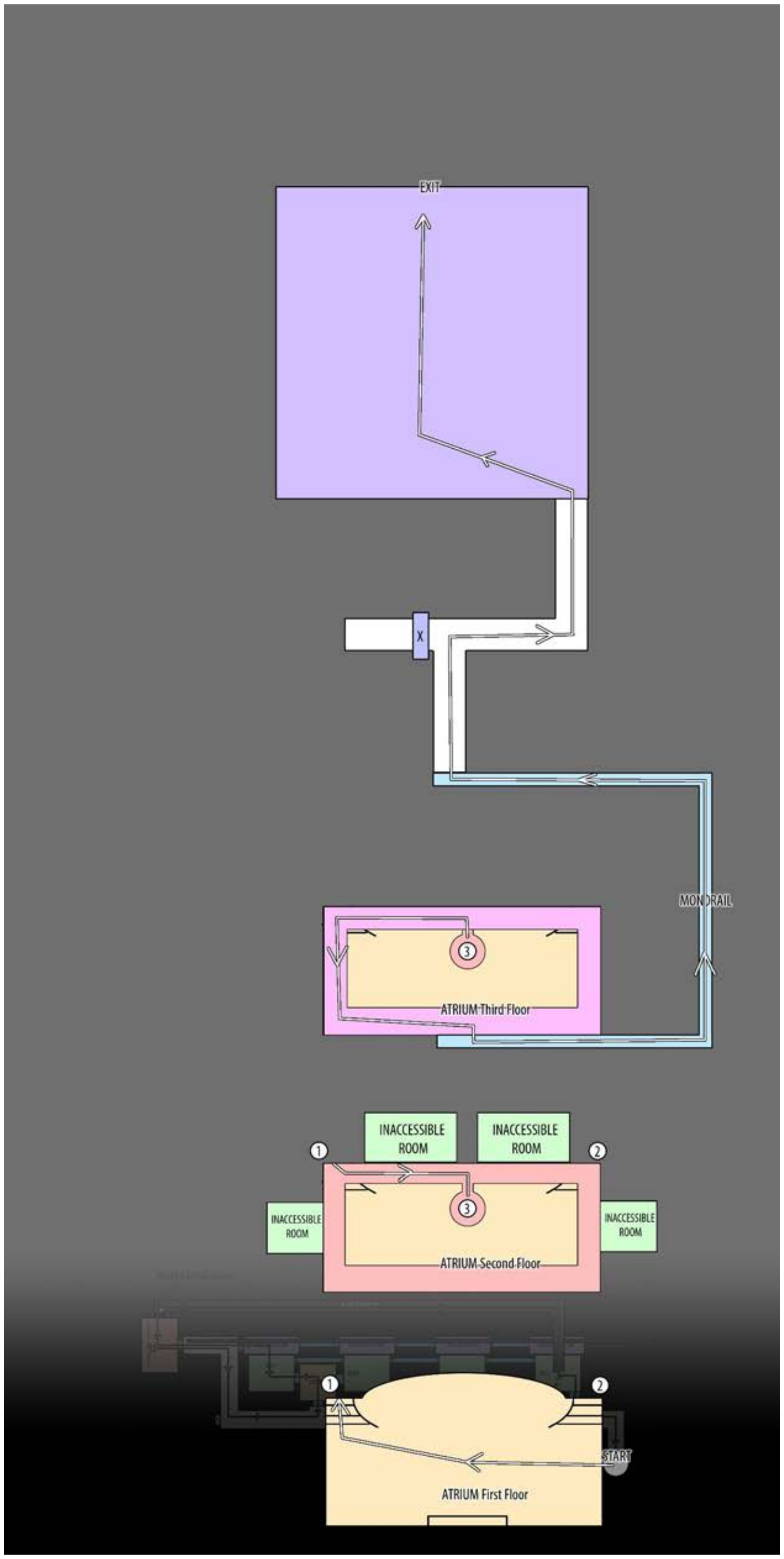


FIGURE 24: CONCEPT FOR THE UPPER FLOORS OF THE FACILITY, WHICH INCLUDED AN ATRIUM AND MONORAIL.

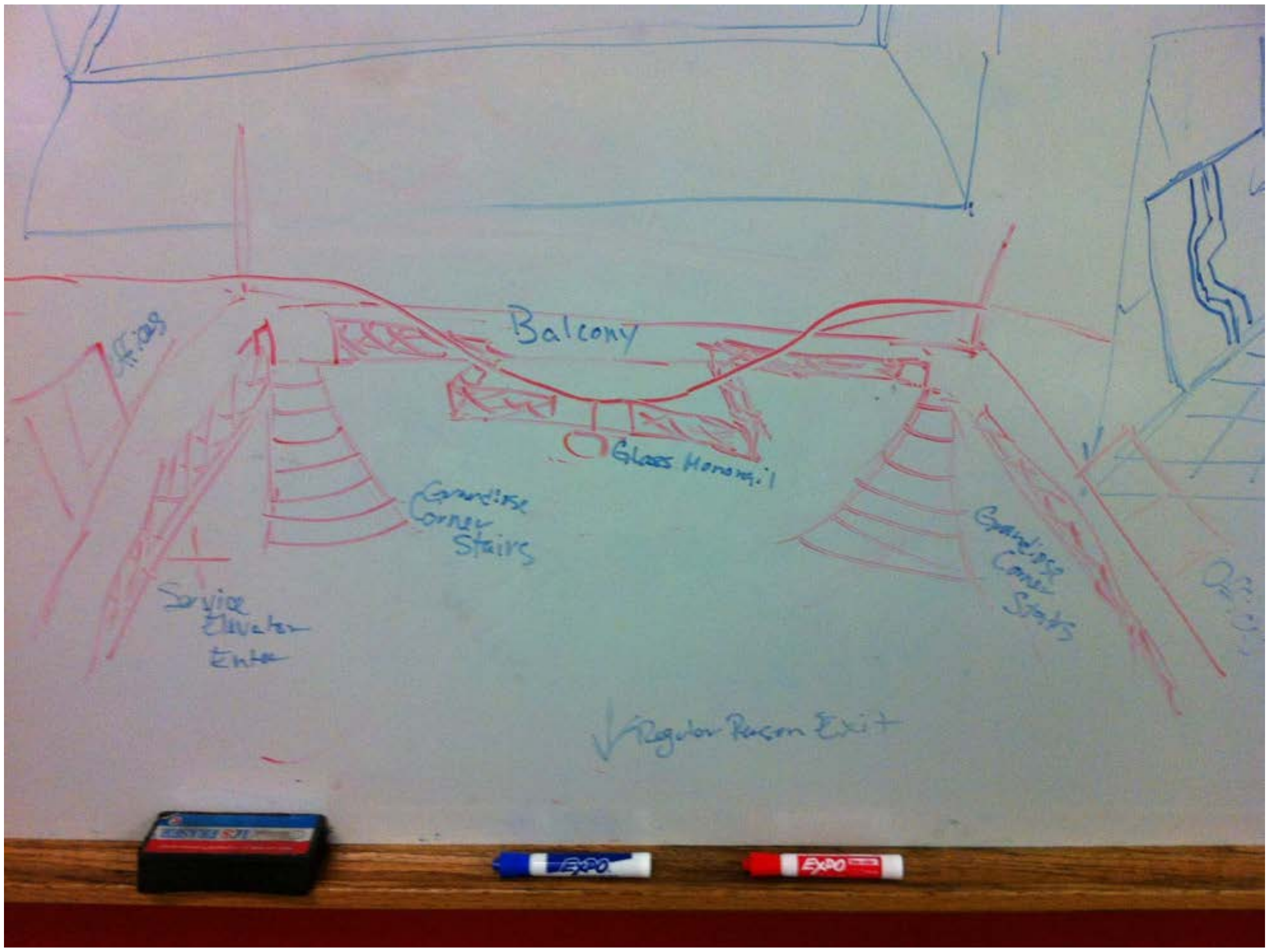


FIGURE 25: EXPANDED CONCEPT ART OF THE ATRIUM IN THE FACILITY.

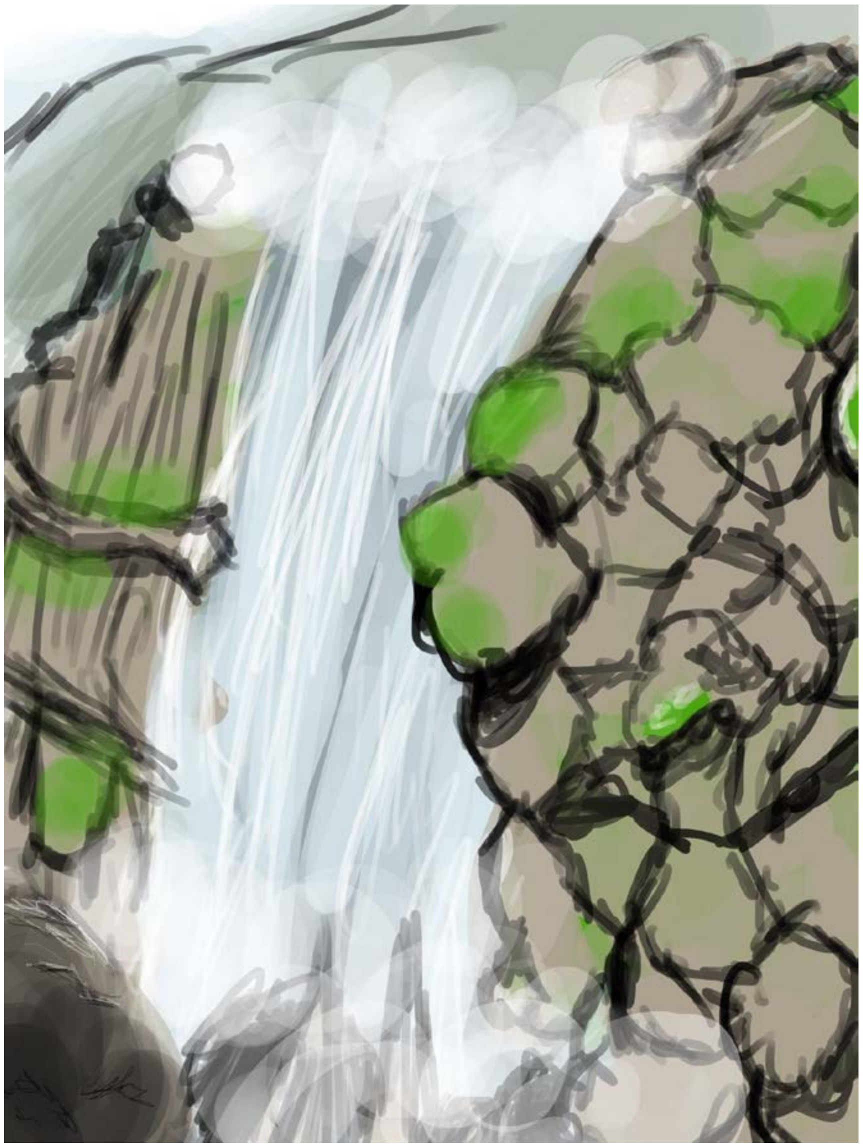


FIGURE 26: CONCEPT OF THE BIODOME'S WATERFALL.

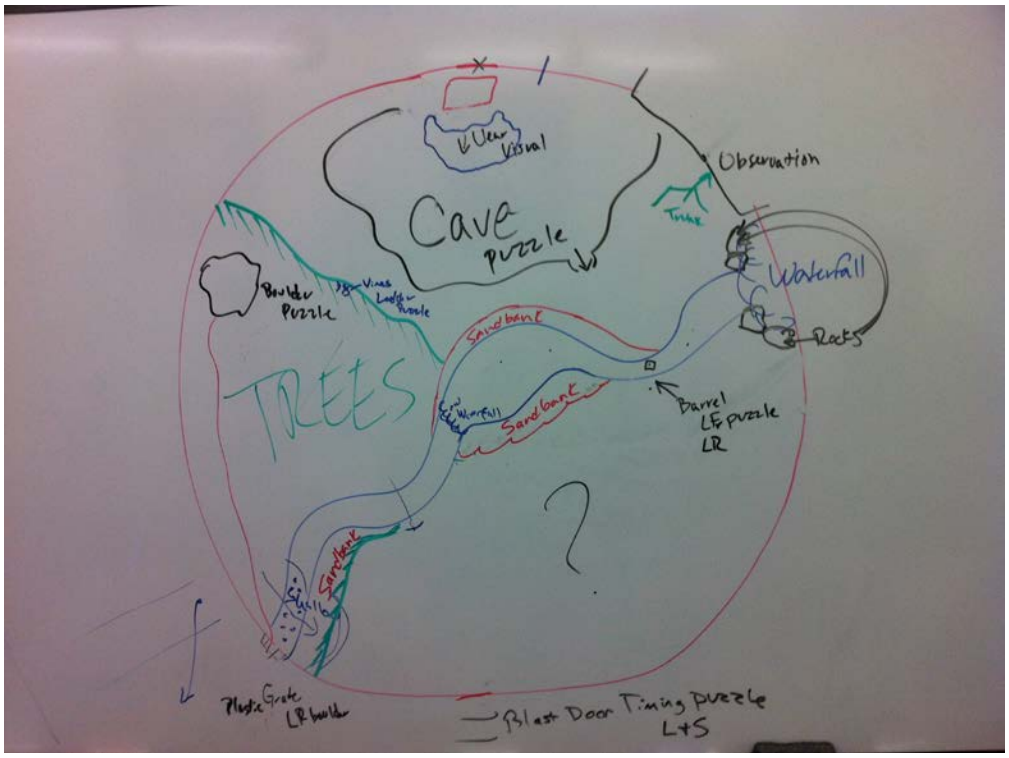


FIGURE 27: VERY EARLY CONCEPT OF THE BIODOME'S LAYOUT.

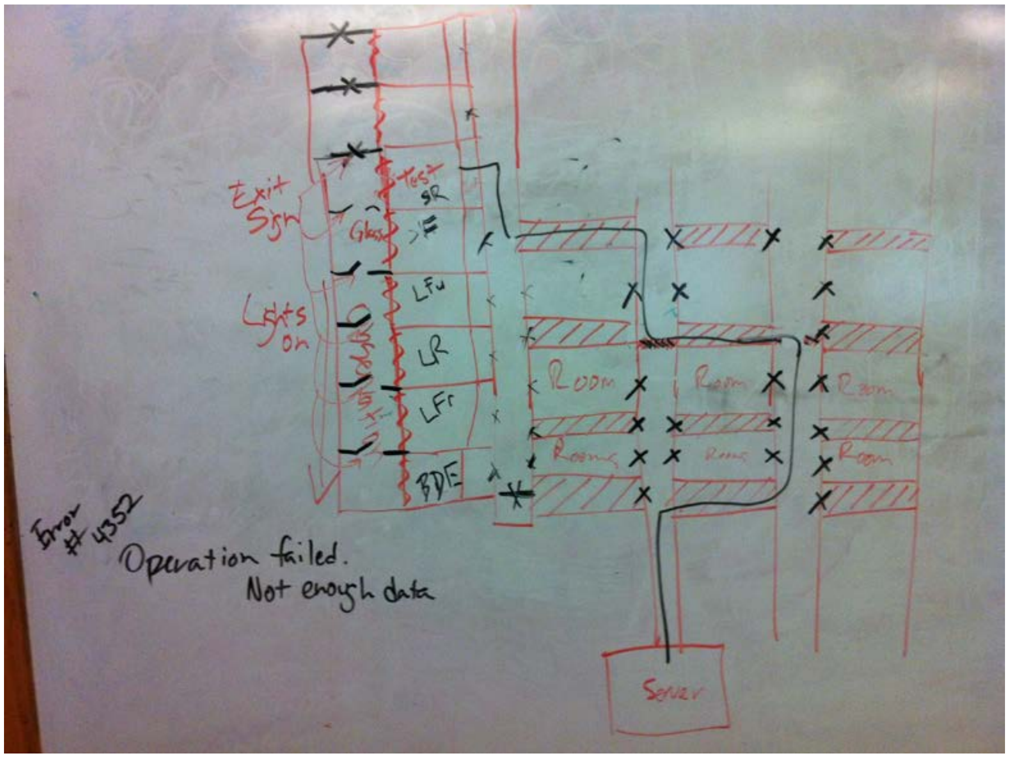


FIGURE 28: AN UPDATED CONCEPT ON THE LEVEL FLOW OF THE FACILITY.

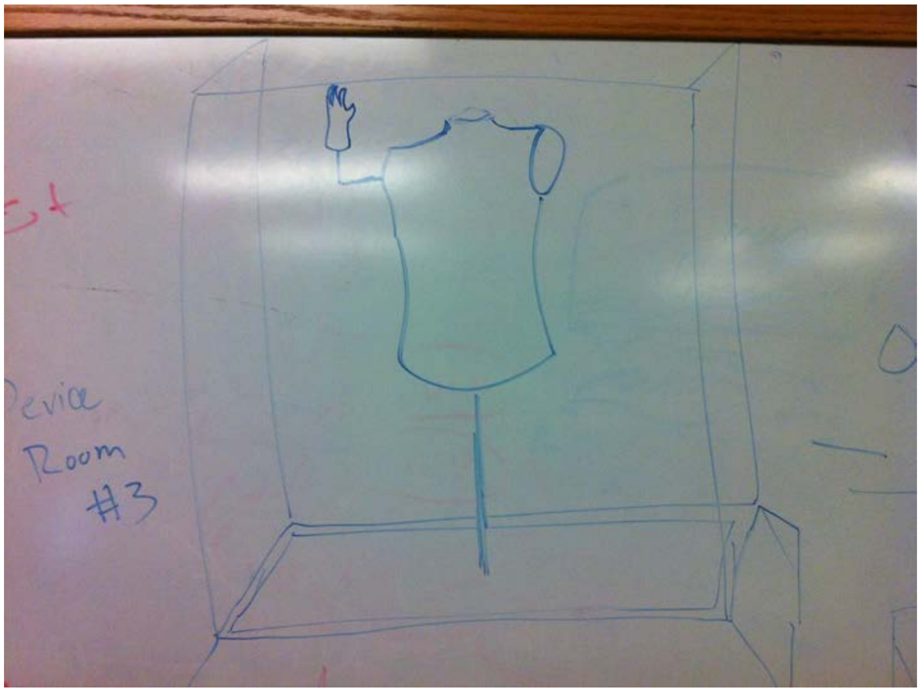


FIGURE 29: CONCEPT OF THE DEVICE PEDESTAL/DISPLAY.

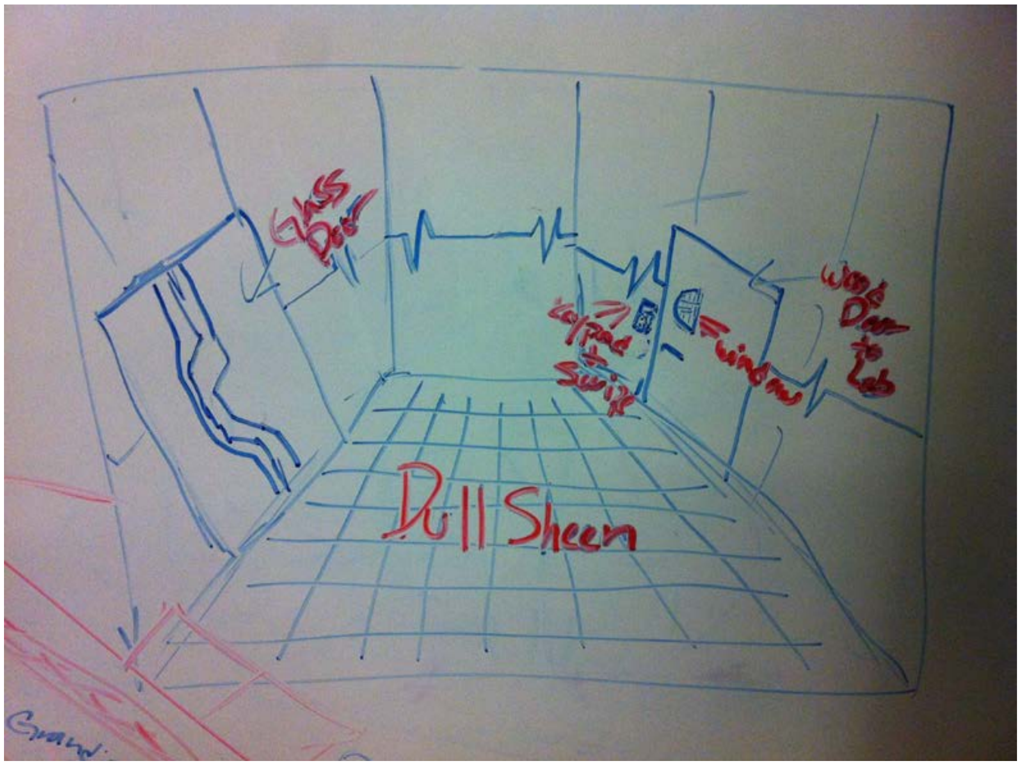


FIGURE 30: CONCEPT OF THE STYLE OF THE FACILITY HALLWAYS.

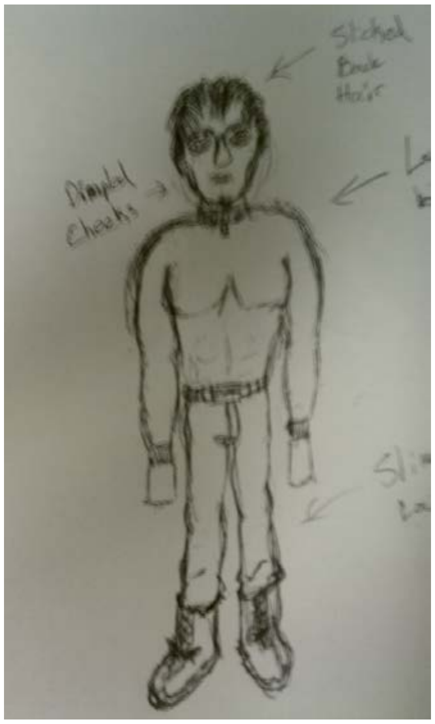


FIGURE 31: JACQUES CONCEPT ART.

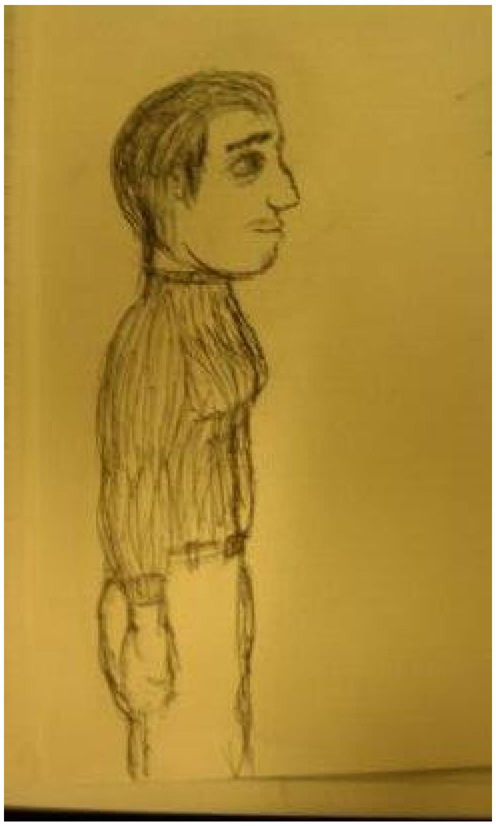


FIGURE 32: ADDITIONAL JACQUES CONCEPT ART.

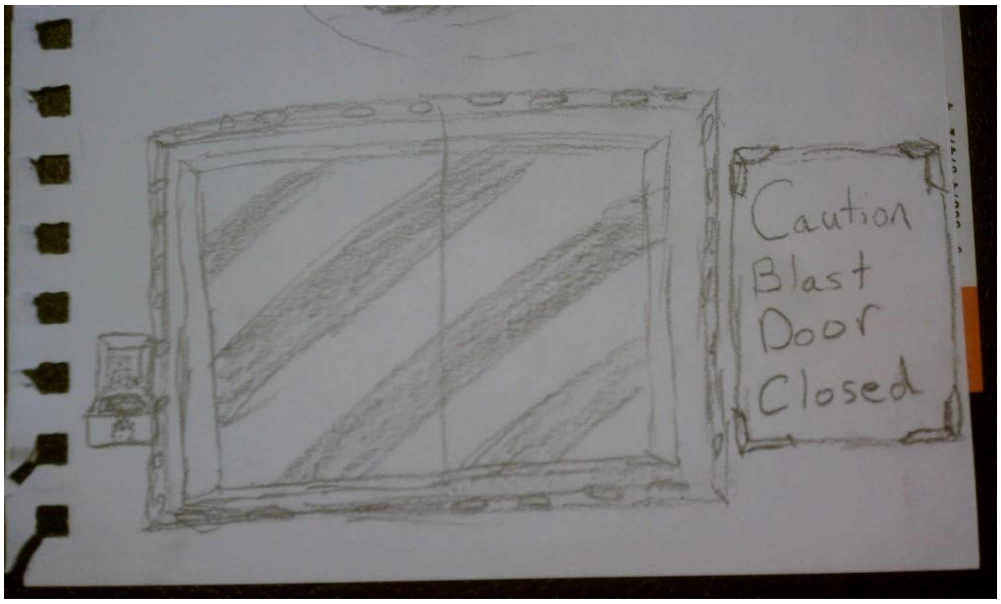


FIGURE 33: FACILITY BLAST DOOR CONCEPT ART.

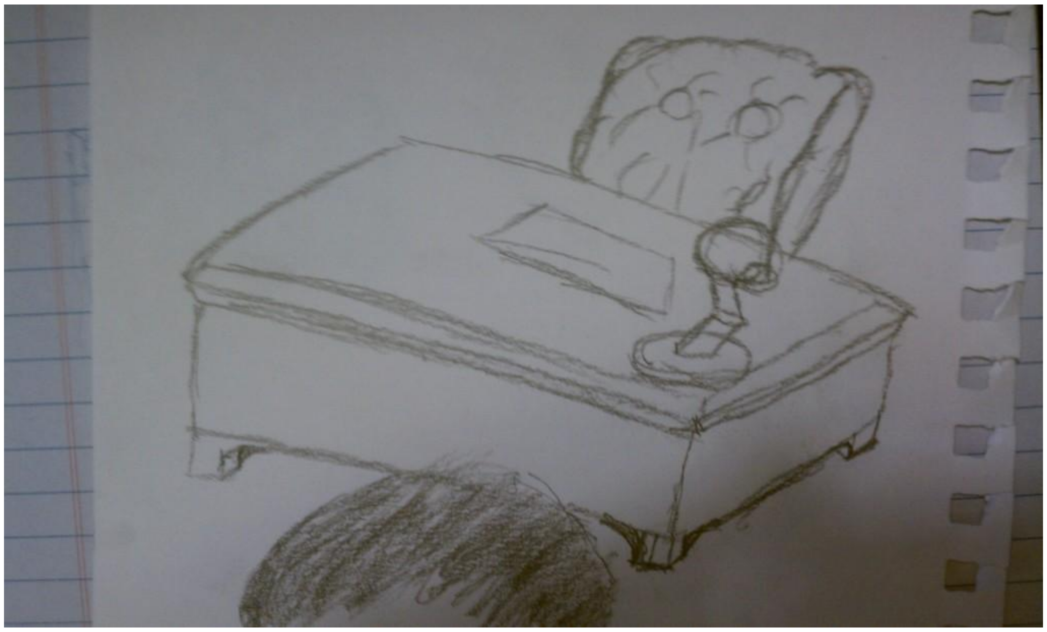


FIGURE 34: FACILITY EXECUTIVE DESK CONCEPT ART.

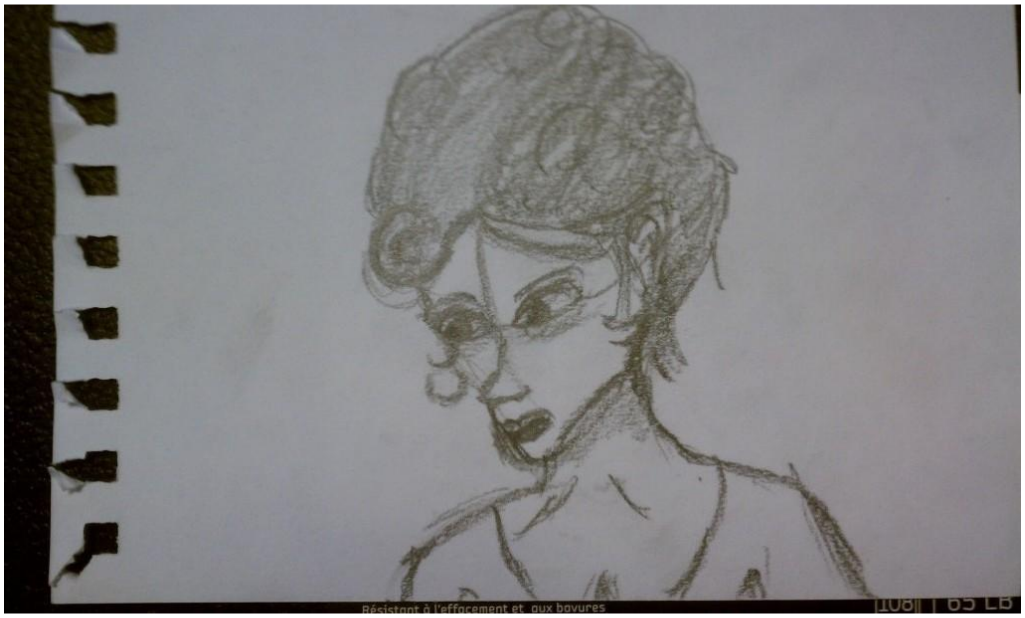


FIGURE 35: CONCEPT ART FOR A SCRAPPED FEMALE CHARACTER.

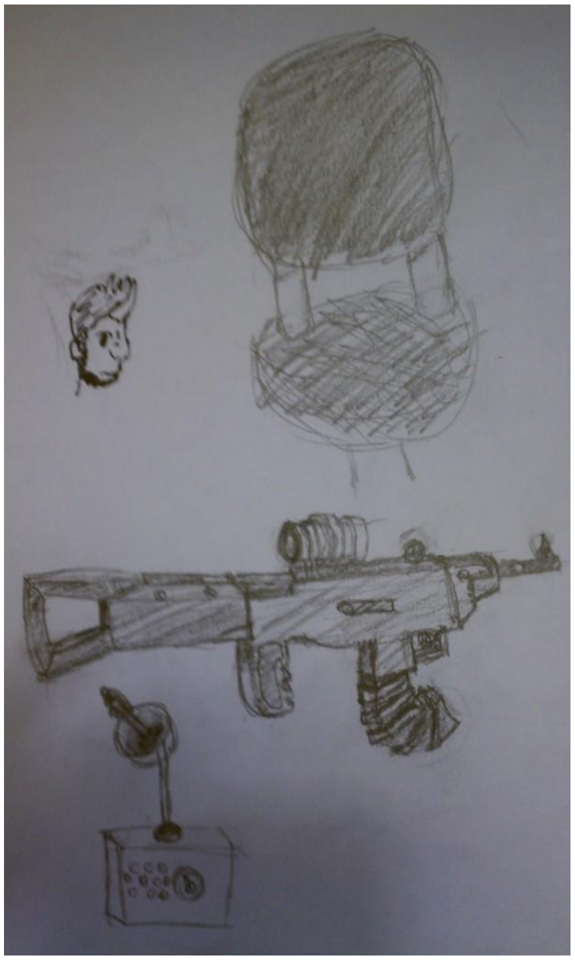


FIGURE 36: CONCEPT ART FOR SCRAPPED ASSETS.

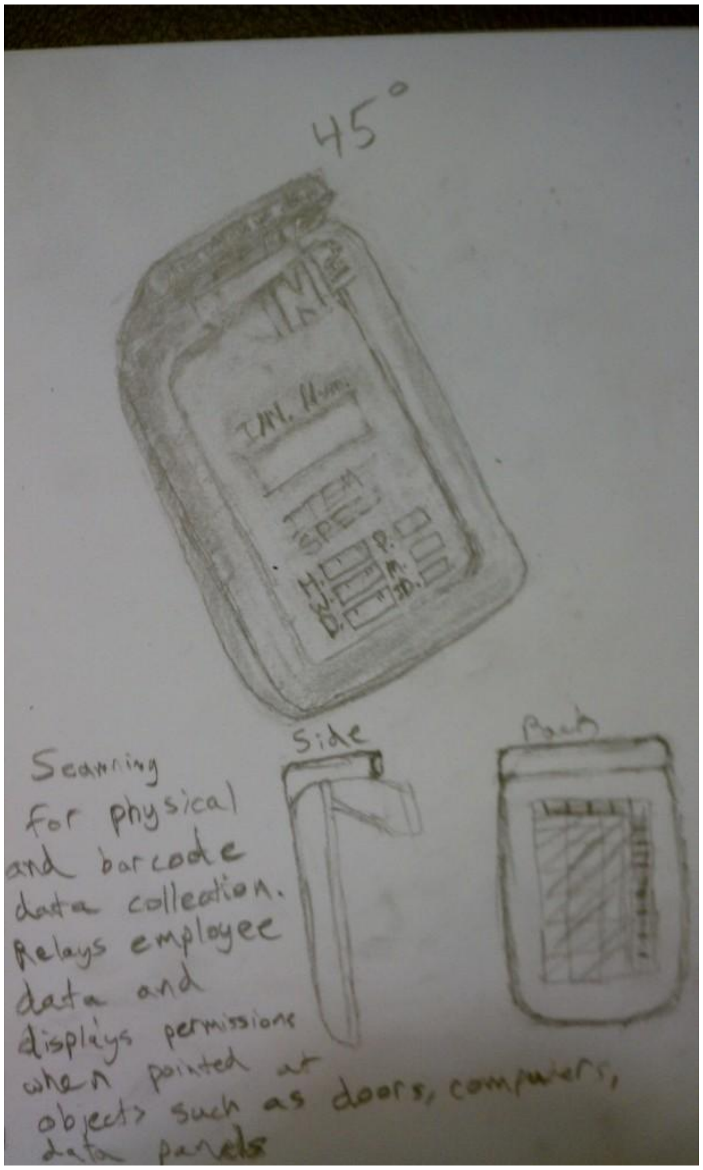
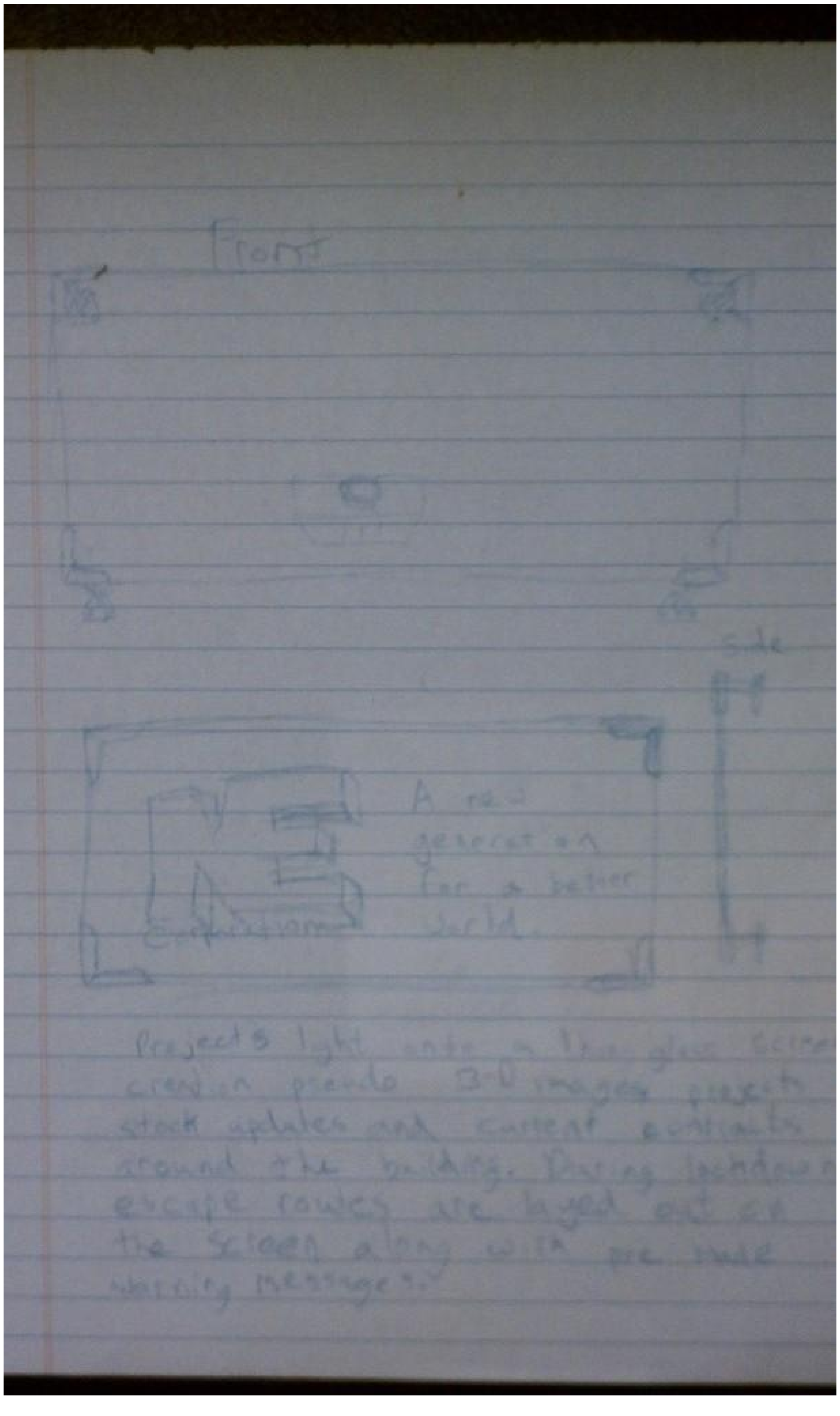

45°
Side
Back
Scanning for physical and barcode data collection. Relays employee data and displays permissions when pointed at objects such as doors, computers, data panels

FIGURE 37: CONCEPT ART FOR SCRAPPED SCAN PAD ASSET.



Projects light onto a thin glass screen creating pseudo 3-D images projects stock updates and current exhibits around the building. During lockdown escape routes are layed out on the screen along with pre made warning messages.

FIGURE 38: CONCEPT ART FOR THE GUI.

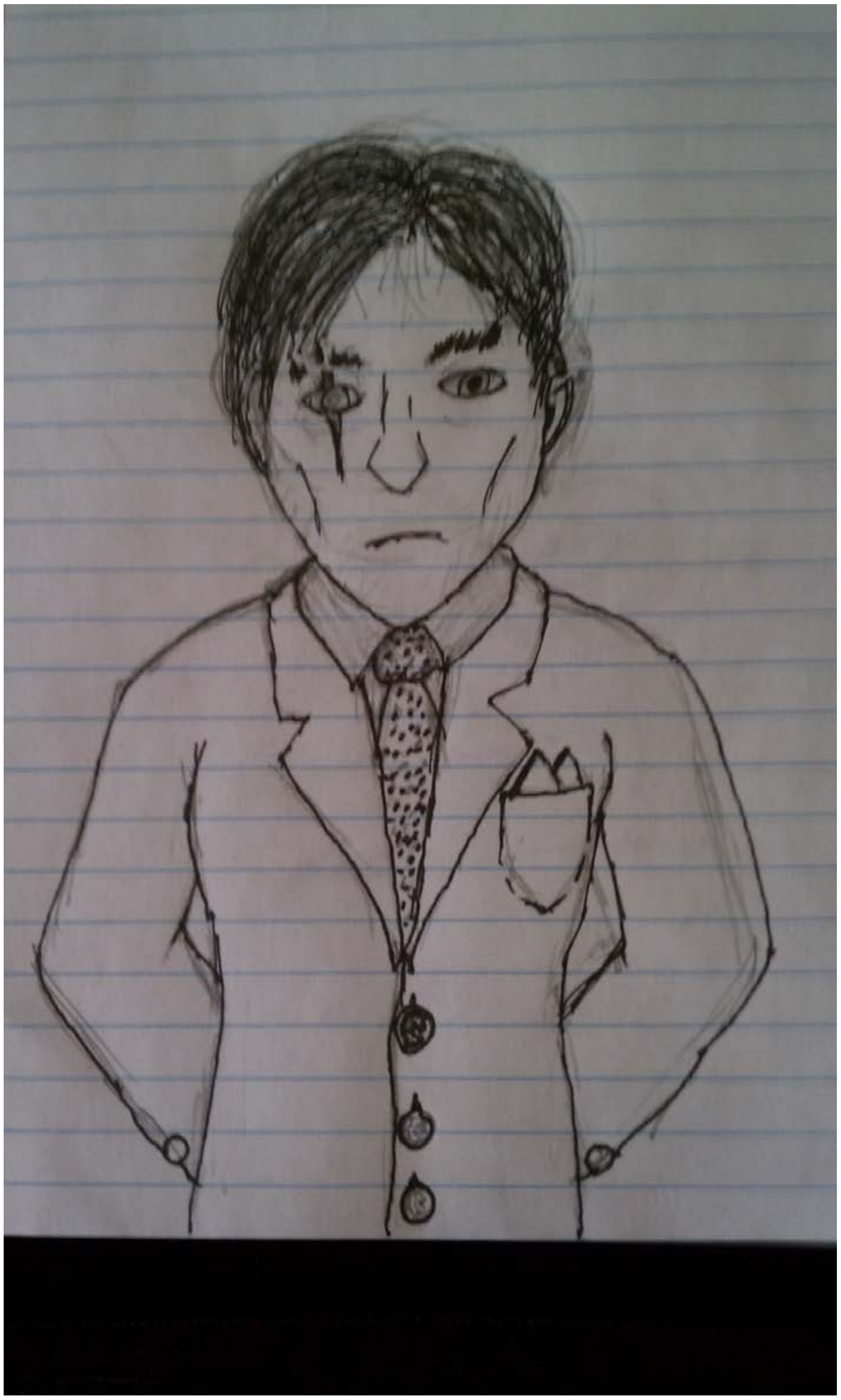


FIGURE 39: CONCEPT ART FOR SCRAPPED VILLAIN.

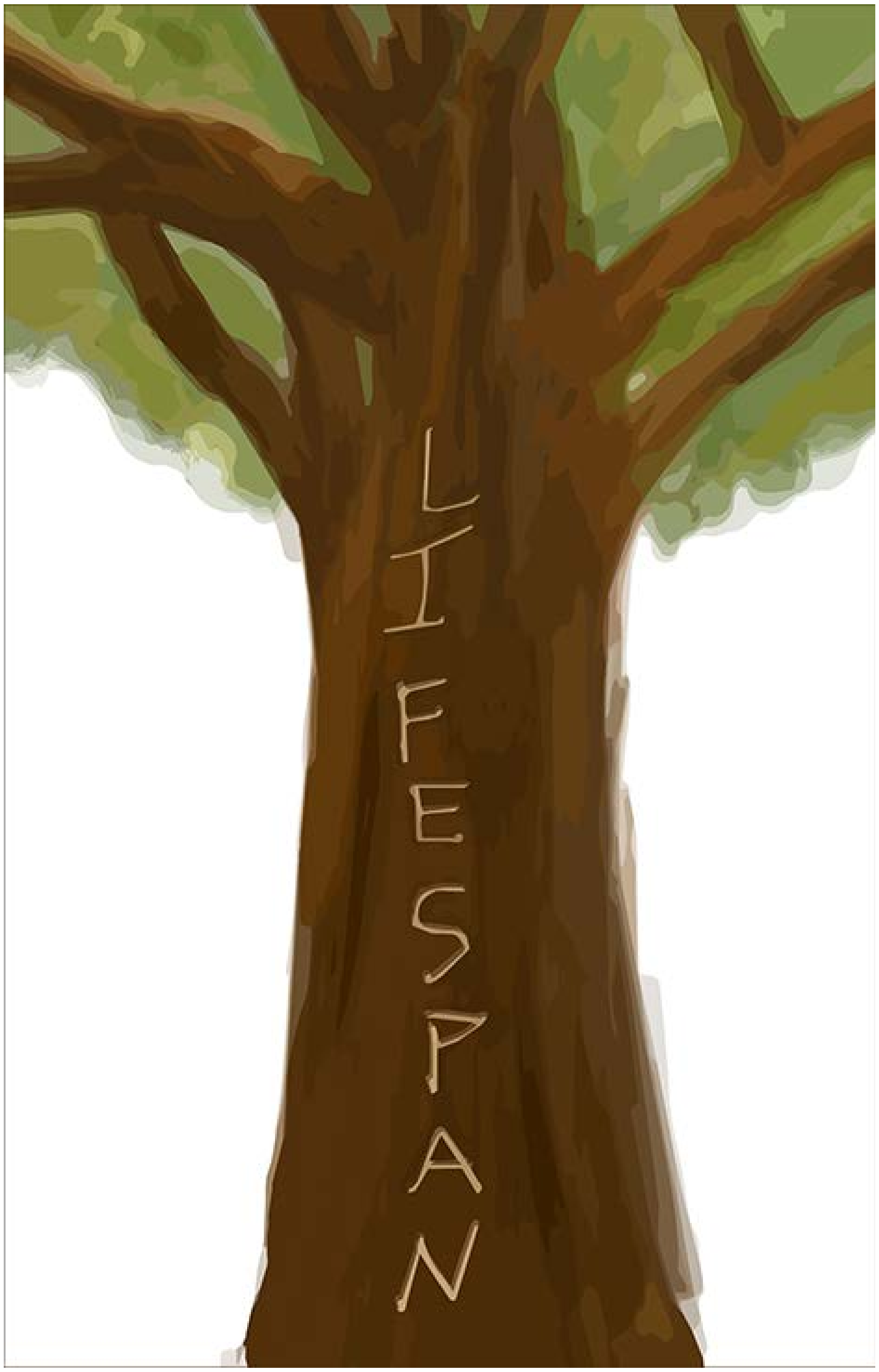


FIGURE 40: LIFESPAN CONCEPT LOGO.

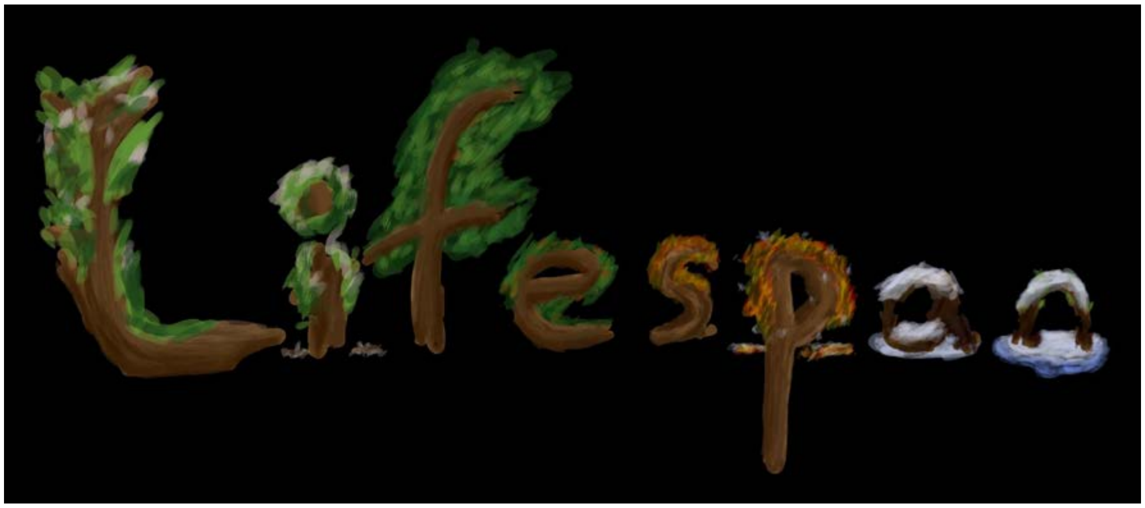


FIGURE 41: LIFESPAN CONCEPT LOGO.

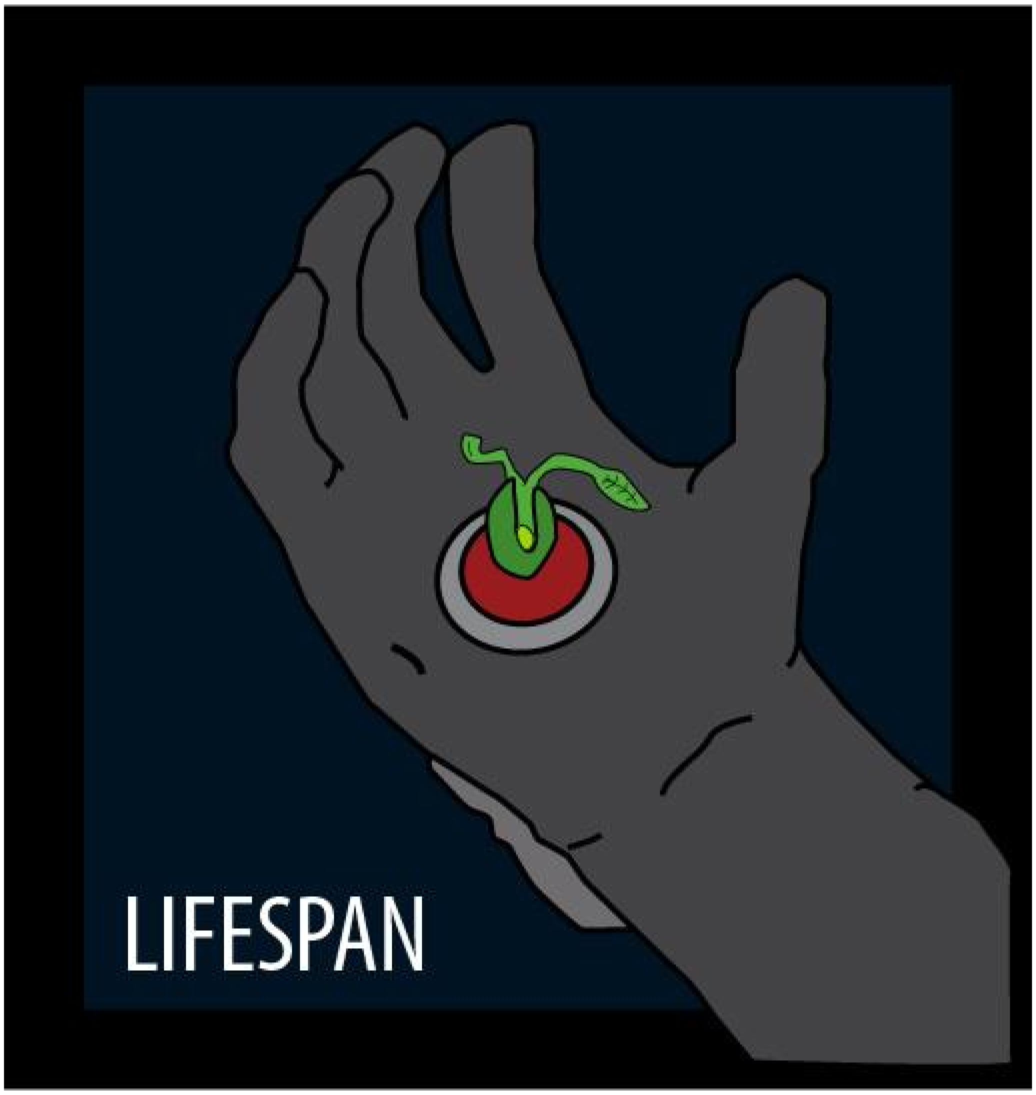


FIGURE 42: LIFESPAN CONCEPT LOGO.

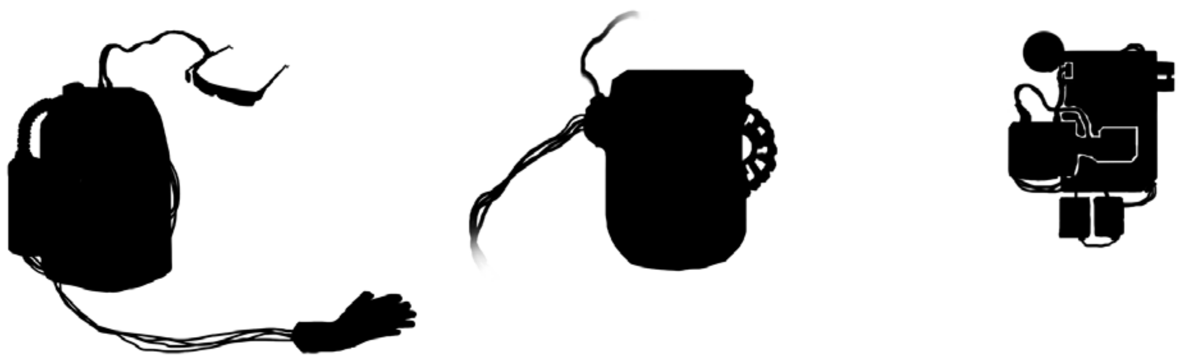


FIGURE 43: CONCEPT ART FOR THE DEVICE BACKPACK.

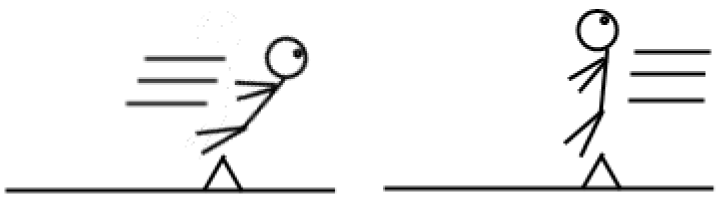


FIGURE 44: LOCATION POWER TRAY ICON CONCEPTS.

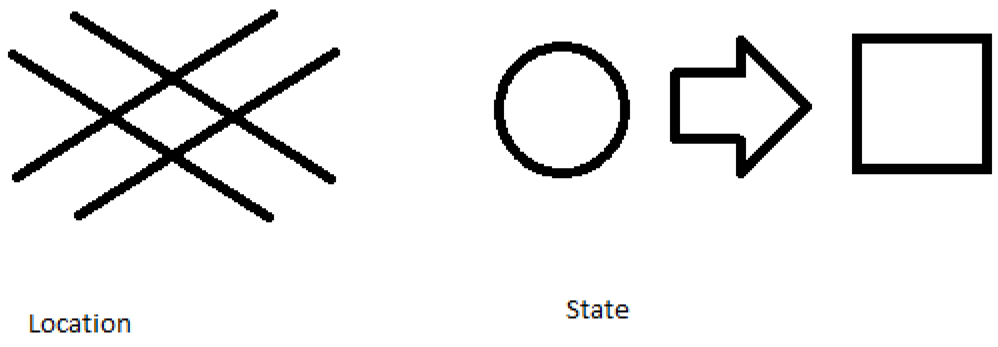


FIGURE 45: STATE POWER TRAY ICON CONCEPTS.

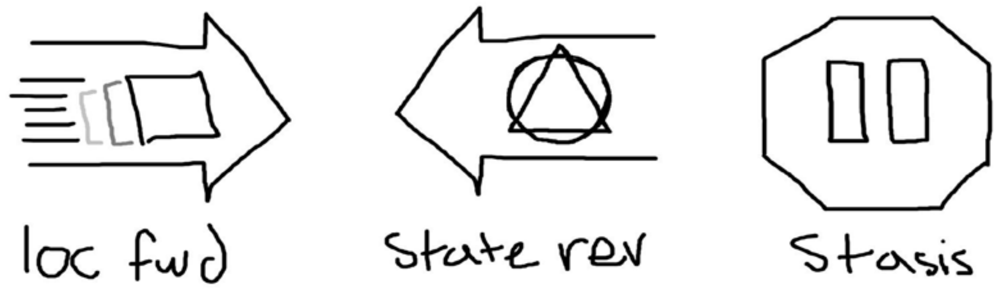


FIGURE 46: POWER TRAY ICON CONCEPTS.

Appendix B: Promotional Materials

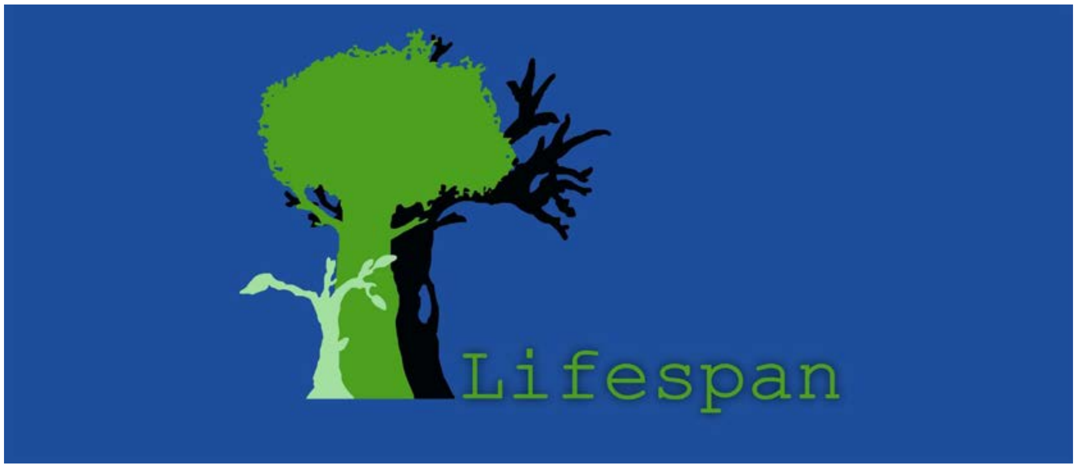


FIGURE 47: LIFESPAN PROMOTIONAL LOGO.

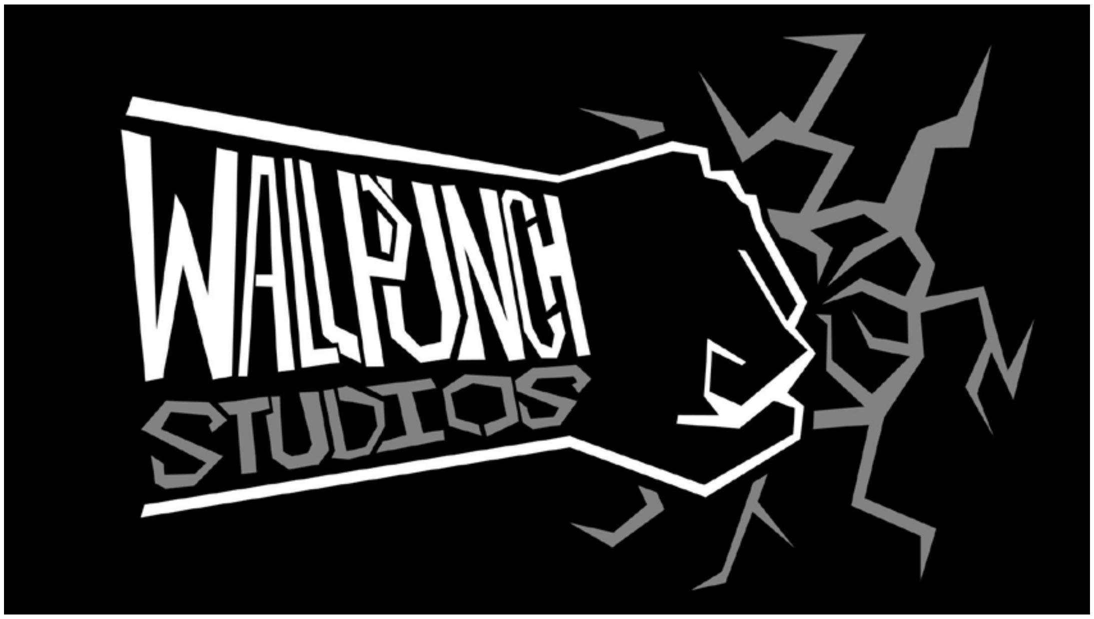


FIGURE 48: WALLPUNCH STUDIOS PROMOTIONAL LOGO.

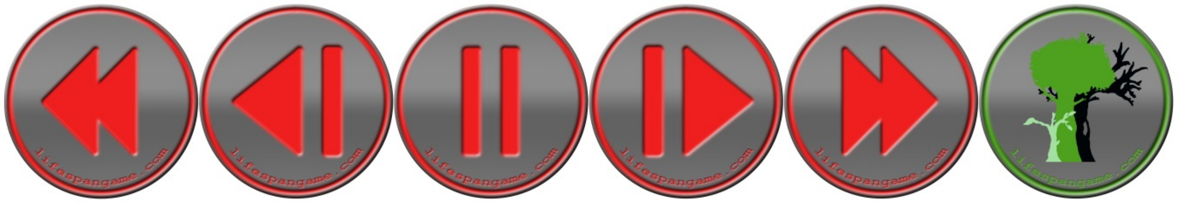


FIGURE 49: PAX EAST LIFESPAN PROMOTIONAL BUTTONS.

APPENDIX C: LEVEL EDITOR DESIGN DOCUMENT

Level Editor Design Document

Developer

Colin Ogren

MQP Team

Michael Grossfeld

Nick Silvia

Matt Tomson

Mike Pelissari

Will Stockinger

Started May 25, 2012

Version 2.0

Overview

The Level Editor is a set of custom **components** and **Inspector** modifications of the Unity **Game Engine** Editor designed to accelerate the creation of scenes in the **Unity Editor**. Its specific purpose is for developers of the *Lifespan* **Major Qualifying Project**, or MQP, team to build puzzles quickly for the computer game *Lifespan*. The interface of the Level Editor exists partially as a modification to the Inspector's Transform component, and partially as a components that constrains the selected **GameObjects**. The controls on the interface call **scripts** written in C# (see-sharp) which call the Unity **API** to activate features in the Level Editor. Such features include snapping objects together, locking properties of objects, and importing from and exporting to **prefabs**.

Concepts

Goals

The completion of the Level Editor is in partial fulfillment of the requirements for the Major Qualifying Project at Worcester Polytechnic Institute. In doing so, the Level Editor will enable developers to build three-dimensional sets easily and more quickly than in the base Unity Editor. Each set will save as a prefab. A prefab is a connected group of three dimensional models, images, sounds, scripts, and settings stored in one file. The Level Editor will also open such prefabs.

The Level Editor will integrate directly into the standard Unity Editor, utilizing the full extent of Unity's scripting API.

Assumptions

The main motivation behind the Level Editor's creation is that the sets created with the Level Editor are intended primarily for a player character in a computer game to interact with. As such, the Level Editor will have to make sure to preserve any interactivity they would have with the player.

Technical Requirements

Hardware Constraints

As the Level Editor will run in Unity 3.5, the user's computer must have the necessary hardware to support Unity. The Unity website specifies the requirements as follows:

- Windows XP with Service Pack 2 or later, Mac OS X with an Intel CPU and Leopard 10.5 or later.
- A graphics card with 64MB of **VRAM** and pixel shaders or four texture units. Almost all modern graphics cards meet this criteria.
- If using **occlusion culling**, the user also needs a **GPU** with Occlusion Query support ("System Requirements," 2012).

Constraints of Unity

As the Level Editor uses the Unity API to carry out the user's actions, the limits imposed by Unity must be considered. Unity can parse scripts in three different languages, Javascript, Boo, and C# ("Scripting Overview," 2012). As the MQP team has decided to use C# for the main project, this language will also use C# in all scripts. The C# language also functions more closely to formal compiled programming languages than Javascript and is more familiar to the developer than Boo which makes it the most suitable candidate for satisfying the Computer Science requirement in the MQP.

Unity also contains an extensive API to write extensions to the Unity Editor's functionality. Classes dealing with **GUI**s and **assets** will prove most critical to the implementation of the Level Editor. Full documentation of the Unity API can be found at http://unity3d.com/support/documentation/ScriptReference/index.html.

Unity Integration

Interface

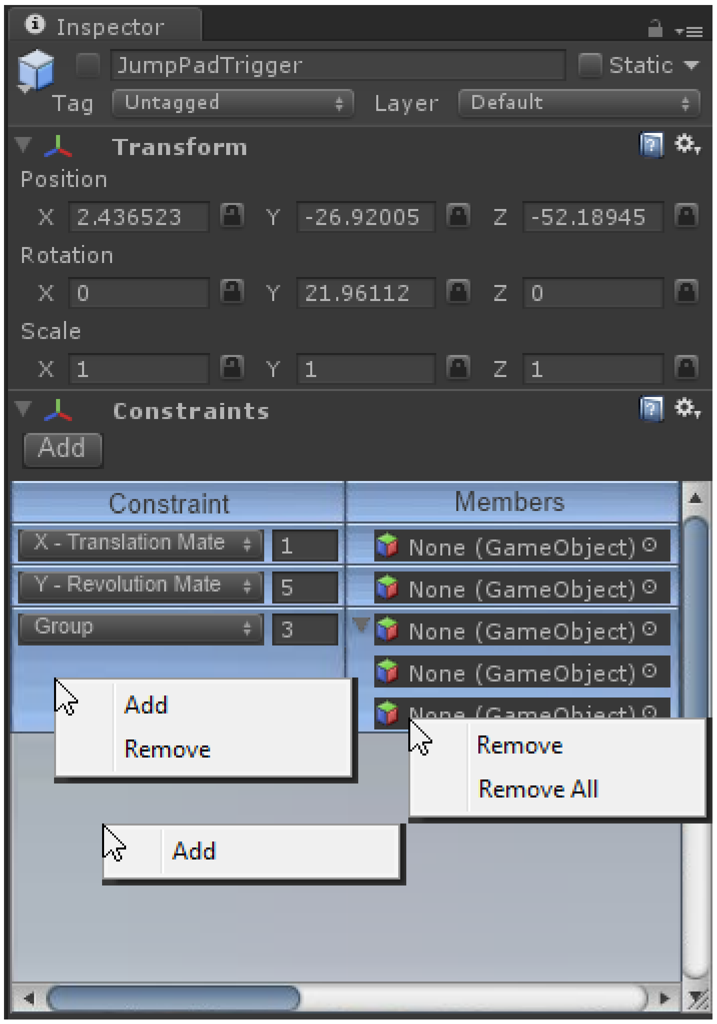
The graphical user interface, or GUI, will draw to the Inspector and constraint component in the Unity Editor. Components will follow the Unity graphical style and color scheme.

Backend

Controls on the interface will activate C# scripts which will in turn make calls to the Unity editor to achieve the necessary functionality.

Specifications

Interface


JumpPadTrigger
Tag: Untagged
Layer: Default

Transform
Position X 2.436523 Y -26.92005 Z -52.18945
Rotation X 0 Y 21.96112 Z 0
Scale X 1 Y 1 Z 1

Constraints

Constraint		Members
X - Translation Mate	1	None (GameObject)
Y - Revolution Mate	5	None (GameObject)
Group	3	None (GameObject)
		None (GameObject)
		None (GameObject)

FIGURE 50: LEVEL EDITOR INTERFACE

The interface exists as a part of the Inspector. The data in the Level Editor reflects the currently selected object. The first section is the modified Transform component for GameObjects, but each value has a button next to it to toggle the lock on the value that allows the user from modifying the value.

The second section is a component that holds a scrollable list of constraints. The context menu allows users to add constraints, and when clicked over a constraint, the menu allows the user to remove it. Each constraint selects from a drop-down list. Depending on the constraint, the numeric input field can accept either a floating-point number or an integer.

For Mating, the input accepts a floating point number as the offset. The GameObject field next to it accepts the GameObject that the currently selected GameObject is mated to. The context menu allows the user to clear the GameObject field.

For Groups, the input field only accepts integers as the number of GameObjects in the group. The number causes the collapsible list of GameObjects in the group to expand or contract as necessary. Note that decreasing the number removes GameObjects from the group starting at the bottom. That is why the context menu allows the user to remove GameObjects in the middle of the list, which automatically decreases the list length. Remove All completely removes the constraint.

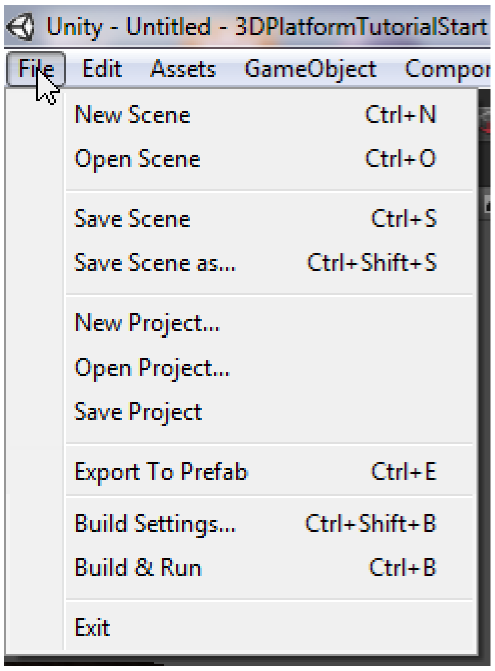


FIGURE 51: LEVEL EDITOR MENU

The File Menu contains an option to export the entire scene as a prefab. This option creates a prefab, but it does not place it in the scene.

Features

GameObject Mating

When the user has a GameObject selected, he or she can then select one of three constraints: mate to x, mate to y, or mate to z. Dragging a second GameObject into the constraint's GameObject field will attempt to move the first selected GameObject along the chosen axis to the second selected GameObject and mate them on one side with a user-specified offset. Dragging the GameObject will translate the GameObject freely in the other two dimensions but when dragging the object in the constrained dimension, the mated GameObject will follow it if possible. Note that mated objects cannot rotate. The exact algorithm is explained below in the Backend section.

The user can also orbit GameObject revolutions around other objects in the x, y, or z axis. The first selected GameObject acts as the revolving GameObject, and the second selected GameObject acts as the reference axis. This allows the first GameObject to revolve around the second GameObject instead of translating along the linear axes. Dragging the mouse to the left will revolve the object clockwise from the user's perspective, and dragging the mouse right will revolve the object counterclockwise from the user's perspective. The user can also specify the radius of the revolution.

GameObject Grouping

Multiple GameObjects can be grouped together. These objects will translate and rotate as a prefab. To group objects, the user selects a GameObject, creates a group constraint and adds other GameObjects to the group from the Projects pane. The user can ungroup a group by removing GameObjects from the group constraint list via the context menu. The group will have its own Level Editor Component displaying the group constraint with its members along with the constraints on the group which are separate from the objects contained within the group.

Position Locking

When the user has selected a GameObject, he or she can select to lock the position of a GameObject along the x, y, and/or z axis, or the user can lock the angle bearing of the GameObject around the x, y, and/or z axis. Dragging the object will move it in the remaining free dimensions but will not move the GameObject from its locked position. In an angle lock, the GameObject can only rotate around the free axes. In a scale lock, the GameObject can only scale in the free axes.

Undo/Redo

A user can undo the creation and removal of constraints and value changes, including the assignment of GameObjects. The user can also redo these actions.

Export to Prefab

The Level Editor can save a scene to a prefab file.

Backend

GameObject Mating

Mating GameObjects requires both GameObjects to possess a transform component. The first GameObject that was selected before choosing the Mate option will attempt to move next to the second selected GameObject. If the first GameObject cannot move, the second GameObject will attempt to move next to the first GameObject. If this is not possible, the user will receive a notification and an error bell will sound. When mating GameObjects, the GameObjects will mate at the nearest point.

When translating mated GameObjects, the selected GameObject will move the second GameObject as it moves.

Creating revolutions flags the first GameObject with a reference to the GameObject containing the axis of revolution. The Level Editor detects an attempt to translate the GameObject and instead revolves it around the axis of revolution with the given radius, either clockwise or counterclockwise, depending on the direction of the mouse.

Constrained GameObjects will daisy-chain to maintain the constraints. The constraints form a network, so for each GameObject, the Level Editor component will use the constraints on the object to determine if the object can move in the necessary direction. The Level Editor will then check the GameObjects tied to the constraints, marking the constraint as checked.

Object Grouping

Grouping GameObjects places them in a prefab and re-imports them into the scene, storing the prefab in a “Groups” folder. The Unity Editor takes care of all the translations and rotations. Ungrouping GameObjects removes the GameObjects from the prefab, destroys the prefab, and creates a new prefab with the remaining GameObjects. If no GameObjects are left to group with the selected GameObject, the constraint is deleted. Adding GameObjects to the group extracts the GameObjects in the group, deletes the old prefab and creates a new prefab with the new GameObject.

Position Locking

Locked GameObjects are flagged as such and any attempts to change the flagged value is ignored in the scene pane or in the Level Editor inspector.

Specify Coordinates and Rotations

This feature simply modifies the GameObjects transform component to reflect the new value after checking for any flags on the values changed.

Undo/Redo

The creation and removal of constraints are logged along with value changes and GameObject assignments. The user's place in the log is marked.

Export to Prefab

The components will all save into a prefab file. If a prefab file exists with the same name, it will ask the user if they want to replace it. The user can cancel the question and choose to rename the file or cancel the save.

Coding Practices

All C# classes will have a commented description of the class's function. All methods will have a commented description of the method's function, including descriptions of parameters and return values. Properties will have a line comment indicating their purpose unless the purpose is self-evident. Also, complex algorithms will also have at least a line comment describing their function.

This project will not use paired programming, but it will use test-driven development. Tests will use NUNIT, a testing software written in C# for C#, using a setup similar to JUnit. Tests will aim to cover one-hundred percent of the code.

Distribution and Installation

The scripts pertaining to running the Level Editor and Unity expansions will get bundled into a **Unity Package**, which users can import into their projects in order to use the Level Editor. Users will have to import the package for each project where they want to use the Level Editor. New versions of the Level Editor will all distribute in Unity Packages. Users will have to clear out the old scripts so the package can import the new files.

Open Issues

The distribution method could take another route with a packaged installer. However, as the developer does not yet have this skill and this is not a priority, this option will come as an afterthought, if at all.

Legal

Scripts will be developed under an educational Unity license. As such, the Level Editor will be free of charge to anyone, but the MQP team owns all rights to all materials. In fulfillment of WPI's MQP requirements, WPI retains a nonexclusive license to distribute the Level Editor scripts and all associated documentation, including this Design Document.

Revisions to this Document

The current version of this document is 2.0 and was drafted on August 2, 2012. Only Colin Ogren is authorized to edit this document, but members of the *Lifespan* MQP team are encouraged to read this document and add comments on the side. Minor revisions will increment the digits after the decimal point, and major revisions will increment the digits before the decimal point. All revisions are listed below in descending order.

Revision 2.0	August 2, 2012
Revision 1.5	June 26, 2012
Revision 1.4	June 19, 2012
Revision 1.2	June 6, 2012
Revision 1.1	June 4, 2012
Revision 1.0	June 4, 2012

Glossary

- Application Programming Interface (API) - A coding collection that provides an abstraction between the computer's kernel and the source code ("Application," 2012).
- asset - A model, texture, audio, script, or other file external to the Unity Engine code that the Unity Engine links to ("3D platformer," p. 9).
- Component - Functionality that a user can add to a GameObject, such as lighting, scripts, sound, etc. ("3D Platformer," p. 9).
- Game Engine - A software development kit that provides base code and an API for common tasks performed in digital games (Ward, 2008).
- GameObject - The basic building block in Unity. A GameObject contains a collection of components that give it functionality ("3D platformer," p. 8).
- Graphical User Interface (GUI) - An interface based on the manipulation of text and image-based controls ("Graphical," 2012).
- Graphics Processing Unit (GPU) - A processor that computes graphical images alongside the computer's main processor ("Graphics," 2012).
- Inspector - A utility of Unity that exposes properties of GameObject components.
- Major Qualifying Project (MQP) - Worcester Polytechnic Institute's capstone project that allows students to demonstrate their knowledge in their major area of study ("Projects program," 2009).
- occlusion culling - The process that determines what parts are visible in an image rendering ("Occlusion culling algorithms," 1999).
- prefab - An asset that serves as a template for a GameObject ("3D platformer," p. 9).
- script - code that defines how GameObjects and their components interact ("3D platformer," p. 6).
- Unity Editor - An asset-centric application for developing computer games with the Unity Game Engine ("3D platform," p. 8).
- Unity Package - A compressed collection of assets for use in Unity projects.
- Video RAM (VRAM) - Memory that is specially used for graphics cards ("VRAM," 2012).

Bibliography

3D platformer tutorial. Retrieved 6/2, 2012, from http://download.unity3d.com/support/resources/files/3DPlatformTutorial.pdf

Application program interface. (2012). Retrieved 6/2, 2012, from http://dictionary.reference.com/browse/Application+Program+Interface

Extending the editor. (2010). Retrieved 6/2, 2012, from http://unity3d.com/support/documentation/Components/gui-ExtendingEditor.html

Graphical user interface. (2012). Retrieved 6/2, 2012, from http://dictionary.reference.com/browse/graphical+user+interface

Graphics processing unit (GPU). (2012). Retrieved 6/2, 2012, from http://www.nvidia.com/object/gpu.html

Occlusion culling algorithms. (1999). Retrieved 6/2, 2012, from http://www.gamasutra.com/view/feature/3394/occlusion_culling_algorithms.php

Projects program. (2009). Retrieved 6/2, 2012, from http://www.wpi.edu/Academics/Projects/

Scripting overview. (2012). Retrieved 6/2, 2012, from http://unity3d.com/support/documentation/ScriptReference/index.html

System requirements. (2012). Retrieved 6/2, 2012, from http://unity3d.com/unity/system-requirements

Unity script reference - GameObject. Retrieved 6/2, 2012, from http://unity3d.com/support/documentation/ScriptReference/GameObject.html

VRAM. (2012). Retrieved 6/3, 2012, from http://www.webopedia.com/TERM/V/VRAM.html

Ward, J. (2008). *What is a game engine?* Retrieved 6/2, 2012, from http://www.gamecareerguide.com/features/529/what_is_a_game_.php