Obscura

IMGD Major Qualifying Project Final Report

John Andrews

Kevin McManus

Calvin Yoon

Contents

Introduction

Abstract

The purpose of this project is to develop a short game displaying the game mechanic of the computer science concept of a pointer abstracted into a virtual world. With emphasis being placed not only on the art and tech of the game, but also the design of the game, allowing for a complete and fun game experience tailored to the previously stated idea in a short amount of game time. The game will be developed using Valve's Source Engine.

A One Sentence Description

A 3D Source Mod story based puzzle game using mechanics similar to C/C++ pointers.

Intended Audience/Platform

The target audience of this game is those who can play Source Mods (own any Valve game such as Half-Life 2, Portal, Left4Dead) and enjoy games involving puzzles. Currently Source Mods can be played on Windows XP, Vista, and Win7 platforms.

About the Game

The Idea

We had just lost one of our artists who had signed on with us from another project before coming up with the idea behind *Obscura*. We had created a Google Wave as a place to throw up any and all ideas, and while we discussed a bunch of ideas, ranging from actual gameplay, stories, and themes, we realized we were going nowhere. Then Kevin came up with the initial concept of using the idea of “pointers” as a gameplay mechanic. After that, ideas for how everything would work just came flowing out of us, all the constraints we would use, what parts of the concept could be transplanted into the game. Now for those unfamiliar: pointers are a way to connect and reference objects in C++, essentially it’s a way to manage data by having a variable, instead of holding data itself, point to another variable, such that any changes done to one are done to both.

We knew from the onset that the mechanics we developed would be best suited to a small puzzle game. Originally we had planned on designing it so that rooms would have multitudes of items and that to solve a puzzle you first had to choose the right items, and then use the mechanics in a way to complete the puzzle. The constraint at that time was that only objects of similar type, shape, or color would be able to work with each other.

While we’ll discuss later the issues of Obscura’s similarities to another short Source puzzle game, we did take cues from Portal concerning how levels and puzzle are designed. An example of this would be that too many objects, especially unnecessary ones, make the puzzle feel too busy and would only confuse the player. At this point the idea of Obscura became more open and akin to the C++ mechanics it was based on: by removing all unnecessary items it allowed players to interact with all objects.

Developing the Idea Further - The Mechanics

Obscura has four core mechanics that allow the player to manipulate objects to solve puzzles. The mechanics are Focus: selecting an object; Transfer: overwriting another object with a selected object; Project: making a copy of the selected object whereby the copy moves just as the selected object does, and Associate, which ties another object to the selected object and, like Projection, where the selected object moves the associated.

How these mechanics translate to the initial idea of C/C++ pointers has remained mostly unchanged in our project. Associate is nearly identical to the original idea - an object references another, and any changes to one happen to the other. Project is an extension of that, the only difference being that a new object is created - the projection - instead of using an already existing object. Transfer is effectively overwriting one item with another - kind of the inverse of Project. Focus was the way we decided to let the player “manage” all the objects/items in game they’re working with. Since you can create chains of associations, projections, and transfers, Focus became our way of clearly defining what you were using the Obscura mechanics with. While the initial idea of the base mechanics was in place (but not finalized - we made several iterations of the mechanics throughout the project), the actual gameplay idea changed a bit even before development.

Each of these mechanics underwent their own changes from the onset, but one of the most notable is the name changes. Focus was originally called Scan, Transfer was at times Overwrite, Switch and for the longest time Replace, Project was initially Copy, and Associate at first was Link. The original names stemmed from C++ concepts, but once we found our story we realized that the names needed to be evocative of the actions they were representing in-game. We felt that the current names would be easier for players to understand, the former being very vague terms. The new names also come from concepts of psychological therapy, tying the names further to our story.

An In Depth Explanation of Each Mechanic

We realized that players would want to make actions on the fly and so would need to have some sort of “inventory” containing the objects they would be working with. Unlike a regular inventory, the objects must remain in world, so we needed a way to store them and let the player select which one they wanted to work with. We briefly considered having a pop-up selection slot like the standard Half-Life weapon select screen. Since this menu of objects would likely have to be static (to make it easier to see the objects in question) it would clutter up the HUD, and as we had no other need for the HUD, we wanted to keep it as clean and empty as possible. We’ll discuss this later in the shaders/visuals section about how we eventually ended up with the glowing outline look of focused objects, but we felt that being able to vaguely see the outlines of objects even through walls was a better fit as a visual aspect of our game.

Transfer hasn’t changed much from the original concept. The idea is that you have one object A. that you have focused on. You then point transfer at object B, and then object A is

moved to object B's position and object B is removed from the game. It is not deleted, rather it is simply moved outside the map. This is done so that the transfer can be undone - if the player "unfocuses" object A, both objects A and B are put back to where they were when the transfer took place. Originally the tool simply deleted object B, but when designing our puzzles we realized the danger of this action - a player could accidentally transfer over an object that they would need to solve the puzzle and (having transferred over the needed object) would have no way of solving the puzzle without restarting. Rather than simply killing the player if he screwed up, we decided to let the player make this mistake and when they realize their error allow them to correct the situation by unfocusing on the object they replaced with, restoring the former state of the world.

Project was originally thought of as a controller - if there was some object you couldn't physically reach but you could Focus, you could create a projection in front of you, which you could use to control the original. After some thought we decided to flip this relationship around, having the original control the projection, so it made more sense in the context of Associate, which it worked very similar to. Having it the previous way also created a disconnect, because transfer and associate both involved actions manipulating objects, with the selected object doing the manipulation. We had originally wanted to make projects appear transparent, but because of shader issues we decided to have a different color glow, which we then also used for associated items, further cementing the parallels between the two.

Arguably our most abstract and our favorite mechanic, Associate was the original idea behind the puzzles we wanted to make that were “real” puzzles; Associate allows you to pick up refrigerators and bounce copiers. Much like projection, the position of object B is dependent on how you moved object A. The weight is also transferred, and though we do use this, its capacity is limited. We had planned on using weight transfer more, but because of time constraints most of the puzzles that heavily used it were cut.

Finding the Story

As we were building a 3D puzzle game, we immediately (and unconsciously) drew inspiration from Valve’s *Portal*. The first story idea we had was that you were a janitor hired to clean up a mostly abandoned research facility. The janitor would be using the facility’s own research to clean up their other, failed, experiments - the primary mechanic here would be the replace ability, reducing entire rooms down to a single item. Beyond the player, there would be three other characters: a security guard who doesn’t trust the player; a HAL 9000-like AI entity, nicknamed “Big Guy” by the security officer, that apparently runs the labs; and finally a small flying robot named Steve (physically and characteristically similar to *Legend of Zelda’s* Navi and *Borderlands* Claptraps), meant to act as the player’s guide and teacher throughout the game.

As the story develops Big Guy turns evil, Steve turns into a spy for him, and you and the guard need to team up to disable them. We quickly realized that the game held too many similarities to *Portal* and *Portal 2*, and having decided this, began brainstorming additional

ideas. At this time, we had already designed all of the puzzles and general level structure, so we had some framework to work with. We met as a group and came up with our second idea: the virtual museum.

With the virtual museum, you were a hacker breaking into the archive - a state of the art computer security system designed using advanced VR tech. The archive exists as a sort of virtual "museum" containing all its necessary data and files as artifacts that most of the users access similar to AR. Your job would be to break into the system, subverting a janitorial program.

The rooms on each floor of the archives, shaped like a museum, contain various items from specific periods throughout history. What the character wants is at the top of the third floor. As the janitor of the first floor, you have the ability to scan items to interact with them, and the subversion you worked up allows you replace things with whatever you want. Using this ability, you navigate through the museum, steal the piece of information you went in to get, and then madly rush out as the system recognizes your presence and attempts to delete you.

While this story worked, we ultimately decided to continue brainstorming. We felt that this storyline followed too closely other heists such as *Inception*, and we wanted something completely original.

The Final Story

After spending a fair amount of time on the previous two ideas, we wanted to generate a large number of ideas in order to more efficiently finalize the story for the game. We began by simply listing out ideas we thought we could use, and then grouping them, and then creating a series of sentences that could serve as the premise of the game. We then went through them one by one and removed them as we decided they were too similar to other games, or movies, or they didn't have the feel we were going for.

Finally, we managed to pare down the list to just a single sentence: "You have amnesia. You're breaking into your own mind to find you. The problem is, you've built up blocks in your own memory that you need to tear down." We all agreed - this would be the basis for the game's story.

Once we decided that this was the story, most other parts fell into place. The mechanics would become psychological defense mechanisms and coping strategies, and other mental techniques. Scan became Focus; Replace became Transfer; Copy became Project; and Link became Associate. Steve, the helpful secondary character, who started as a small friendly robot, became a doctor and therapist, guiding the character through his own memories. We decided to make the doctor a woman, to give the characters more variety.

We also decided not only that the main character and the doctor would be the only two characters, but also that the doctor would, level by level, say less and less, so that by the fourth level or so, the main character would be alone. This allowed the player and the main character to learn and discover the character's history together.

We decided that we wanted the main character to have a dark story, and that at the end there would be a horrible revelation about who he had been. We quickly decided that this horrible revelation would be that he had killed his family, as this would explain why they weren't

around. We also felt that this could act as a potential reason as to how he lost his memory, although we wanted to keep that explicitly unsaid within the game to add a sense of mystery. In order to justify the character's murder of his family, from a character standpoint, we decided to make him an alcoholic, which we would reveal to the player in the penultimate level.

For the doctor, we wanted her to help the character out as he progressed through his memories, but we also wanted her to have an ulterior motive - in the final scene, she tells an aide, not visible to the player, to "Mark down trial 3 as a failure." This line could suggest multiple things - is the player's experience the third trial of the drug, is this the player's third time experiencing his memories. We wanted to leave this explicitly vague so that the player could give any meaning to the ending.

Designing the Levels

We originally designed the game with seven levels. Early on, we decided to have each mechanic be introduced on its own level, and then have a harder version. Except for the final level, the odd numbered levels (1, 3, and 5) would each present a new mechanic to the player: Transfer (with Focus), Project, and Associate respectively, and provide them with some simple puzzles explaining the functionality of each mechanic. The levels immediately following these (2, 4, and 6) would have harder puzzles using those mechanics. Level 7 would incorporate all mechanics together in the most challenging puzzles. Early puzzles are very simple and linear, but towards the end the difficulty increases, requiring you to fully concentrate on how to solve the puzzle.

Mid-way through development we decided that the puzzles in Level 6 and 7 were either too vague, or too "geeky" (meaning that most people wouldn't understand how to solve them). We combined the two levels using the stronger parts from each into one level, leaving us with 6 total, as well as an introduction scene and a closing scene.

The puzzles for the introductory levels changed very little from when we initially drew them out on graph paper to when they were implemented in-game. The remaining three levels underwent major changes, from discussions with our advisor to “walking through” each level to find the places were a player could break a puzzle and not be able to advance in the game. We also had to change puzzles because some were created using the original projection mechanic, when the projection acted like the controller.

Besides puzzle changes, some other key changes to level design were implemented during the course of development. Lighting and effects for gaps (restoring the floor) required a fair amount of tweaking, since most lights used in Source games (and our mod) are static, with the lighting on objects and walls baked in. We also made changes to the levels based on feedback from initial playtesting sessions. Playtesting also forced us to change a variety of things in each level based on what players could do that they should/shouldn’t have been able to do.

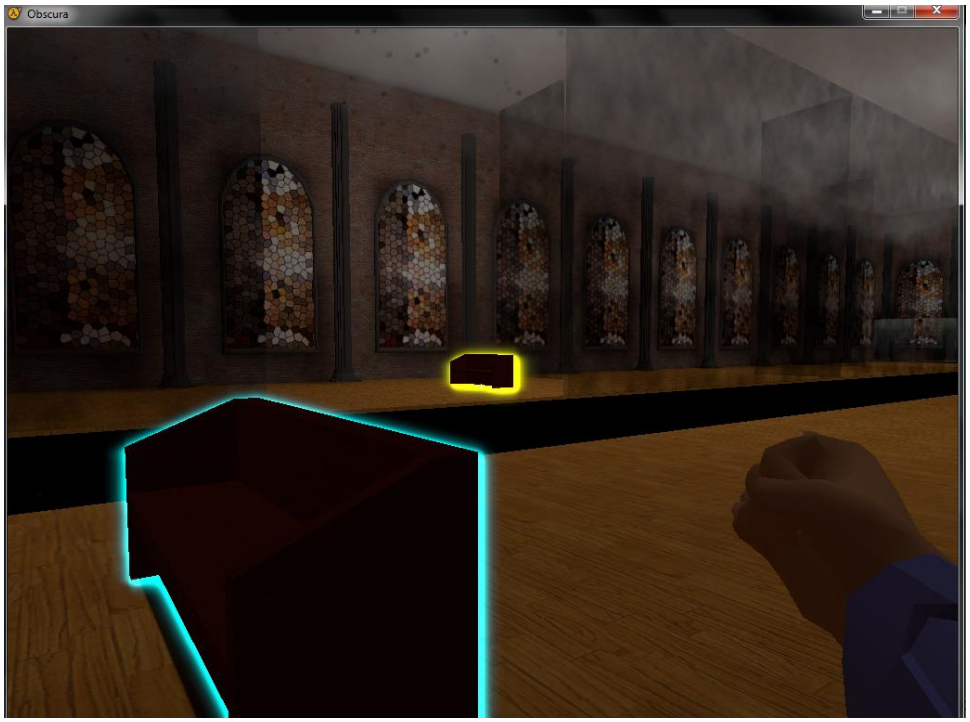
The maps for both the current levels and previous iterations are included in the appendices, below are screens from each level showing either a key part of the level or a key feature of the game.

Below is a screen from Level 1, showing off the Memory Blocks that stop a player from progressing further in his mind. On the left the white halo is a sync zone where a players mind is broken and must be repaired by placing the corresponding item within it. Lastly we can see the visual aspect of “flat objects” which are there to provide clues about the room but to not confuse the player are not interactive objects.

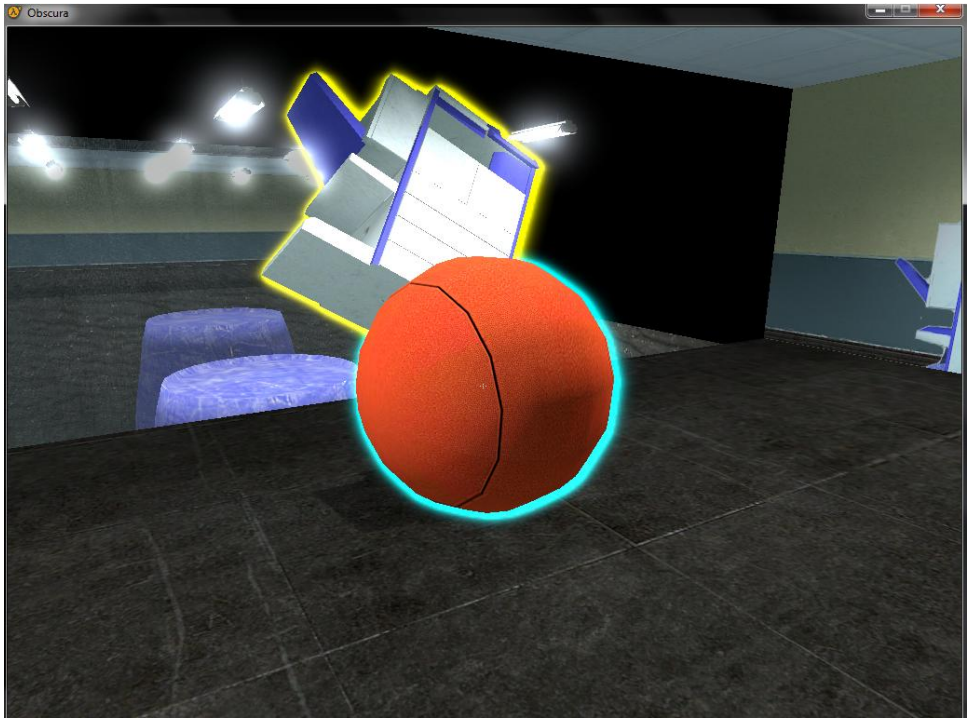

Obscura


Obscura

Above is Level 2 (The School) with the transfer mechanic selected; also shown is the glow surrounding scanned/selected objects. Below on Level 3, we can see the mazelike structure created to show off Projection. You can see the selected object in cyan and the created projection in yellow.



Above we can see the hospital from Level 4, with the final puzzle of 2 baby beds that require both the original and projection to solve the puzzle. Below we see a screen from Level 5 showing the Associated mechanic with the ball selected and the copy machine associated. You can also see that the glow around associated objects is the same as projections.



The following two images are from Level 6, the first showing off a look at one of the rooms, at one end you can see sand with a seesaw and overturned buckets that you must use to navigate the room to the other side. On the other side you can make out the bricks from the church with a stained glass window on the ceiling. This and other examples were done to display the dissonance of the last level with all of his previous memories colliding into one another. The final screen is a glimpse of the final room after the long dark tunnel. This is where we end our exploits of his subconscious with the character waking up from the shock from remembering that he killed his family.


Obscura


Obscura

Developing the Game - The Code

Working with the Source Engine

The Source Engine, much like any other engine, has both advantages and disadvantages in terms of what the engine does well and what it doesn't. For example, building levels and using triggers are handled by Hammer, leaving small changes like object hierarchies outside of the engine. Another benefit of the Source engine is the ability to connect scripting languages like Python or Lua to the enigne, allowing you to control objects on a more modular basis. This was essential to overcoming our problem of activating objects that needed their own actions. Aside from the code for the glow effect and Lua hooks, most of our code was adding our own weapons, and most important modifying physics props. Most of our issues with the actual code of the Source Engine were dealing with physics interactions.

This isn't to say that Source doesn't have problems with its codebase. First and foremost, there is no identifiable organization to the files for the source engine; most things are simply put in a giant "source" folder (this is literally how they decided on the engine's name). This can make finding relevant files very difficult, especially when we needed to debug things. Another glaring issue with the current release of the Source Engine is the shader implementation. Compiling shaders for use in a mod was quite frankly broken for all but the lucky, and we eventually abandoned using them.

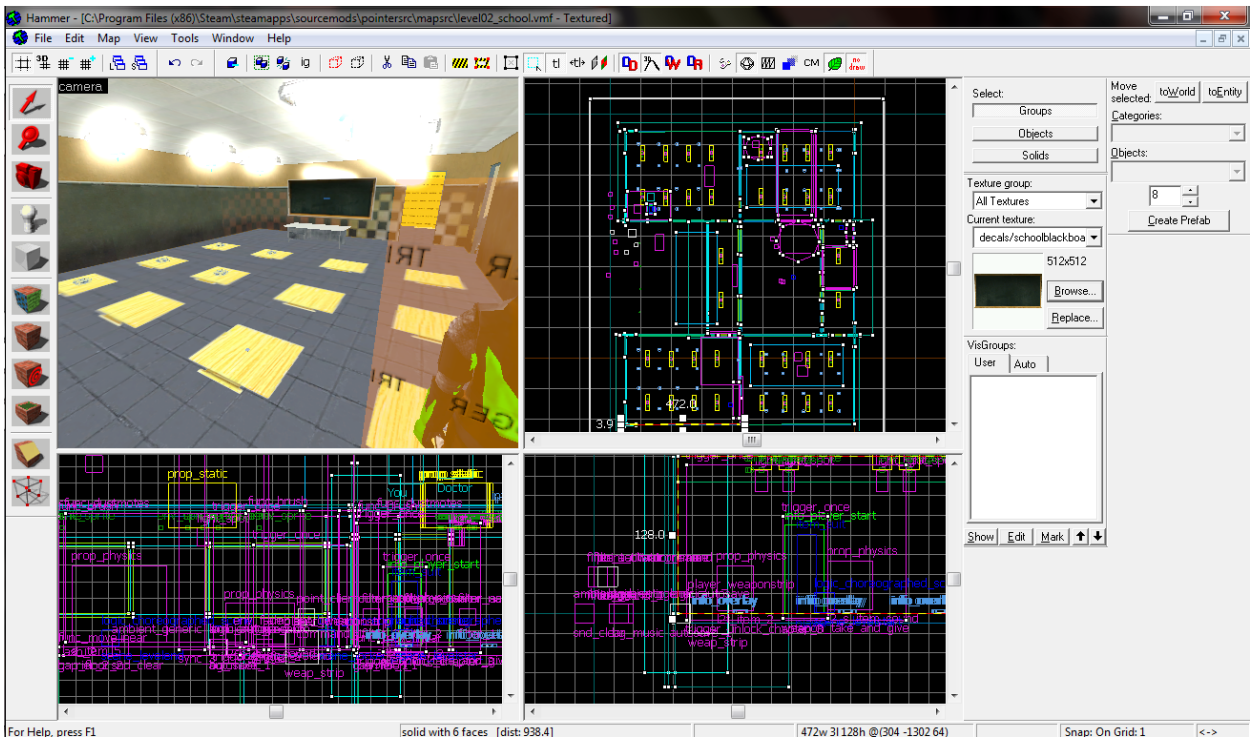
Compared to other engines we've used, the Source Engine wasn't a bad experience and I think we would all be willing to work in Source again.

Working with Hammer

Included with the Source Development Kit is their map editor, Hammer. Hammer is a very powerful tool, allowing you to not only create the physical world and objects within the world, but triggers. Triggers allow you to play sounds, move objects, display text, saving, and changing levels. The gaps and sync zones in our game were only possible because of extensive usage of triggers and logic entities. Working in Hammer was an enjoyable experience, since things could be changed and modified so easily. Since you wouldn't have to compile the engine/game code to change things for a map, development of a map was very easy for testing purposes.

Below is an example of the Hammer interface showing Level 2. The left sidebar contains all of the tools you can use in Hammer for creating entities, selecting objects and applying textures. In the wireframe views the blue lines represent the parts of the world that are walls/floors. The purple lines represent entities ranging from triggers, gaps, blocks, sync zones, etc. The giant white box is meant to contain everything in the map, so that no entity touches the outside of the map; this is a Source requirement to prevent leaks in map files. The small squares outside of the map but within the main box are entities that control commands, logic functions, or filters to only let certain objects interact with sync zones.

In the shaded perspective mode which you can move in using WASD and arrows (for camera), you can see a bunch of items. There is foremost the player, surrounded by a trigger which we use for things like unlocking the chapter, giving the mechanics to the player, or starting the dialogue for level start. You can also see the textures for the walls and floors, as well as the lighting and glow from the fluorescent lights. Finally you can see the two types of objects: the physics prop teacher desk which the player can interact with, and the school desk decals.



Blocks were made by having a glass wall that would start below the world and move up on level start so as to not break the lighting. This is similar to the trick we had to use for the gaps that exist in the level by having them start where they would be when the puzzle is solved, in order for the lighting to bake correctly, and then when the level loaded immediately move them down to their “broken” state. Finally we had to use “block light” entities to repair some situations where light would bleed through walls.

Adding Scripting with Lua

As we designed the puzzles that would become the levels, we realized that we would need functionality in place that would allow us to manipulate items based on character action, or intrinsic properties. For example, one of the first puzzles we brainstormed (that we decided not to use) involved a duck that could glide, and a monkey that could climb. Functionality that was included in the game include small cars that would drive when you use them, and balls that you could throw.

However, we did not want to have to handle any of this functionality within the C++ code. Instead, we chose to implement these features through a higher-level scripting language. The primary reason behind this decision was the freedom this would allow designing levels - we or anyone else creating levels would not need to modify the source code in order to create new puzzles.

Having chosen to embed scripting within Source, we then needed to decide which language to embed. On Valve's Developer Wiki, options exist to extend the engine by adding either Lua or Python. Initially, we attempted to add the Python, as some of us had experience with the language. Ultimately, we decided to go with Lua, as there was less overhead involved within the engine in comparison to the Python system.

Due to the fact that we could not modify Hammer to add scripting hooks, we created a plain text file, mapping objects within Hammer (by name) with paths to scripts. This allowed us to not only customize the scripts for objects on a per-object basis, but if two objects needed the same functionality, a single script could be used for both.

There were two separate stages in the design of the actual scripting mechanism. Initially, whenever an item was activated (the Q key), the entire script would run from top to bottom. Over time, however, it was realized that this setup was not enough, and the system was redesigned so that there were several callback functions that each script would have to implement - one for the object spawn, one for every think function of the item (calls think() roughly every 0.1 seconds), and one each for when the item was used (the E key) and activated (the Q key). This reorganization allows for a greater level of customization in terms of what the scripts are capable of doing. Additionally, a number of getters and setters for the item were created - setting the position, the angles and whether or not it's bouncy. As we developed the levels, other functions were added as we needed them.

Overall, using Lua as a supplement to the C++ code was successful. It allowed us to quickly create and implement puzzles without needing to change the code.

Shaders

We originally planned for our game to have three shaders. The first was a glow effect - focused objects would glow a certain color, and the selected object would glow a separate, more vibrant color to distinguish it. This effect would allow us to display to the player the items they have focused on without needing to put any information on the HUD. The second shader we planned on using was a transparency effect, used for projections. We wanted projections to appear shadowy and hazy, and allow players to see though them, to make it clear that they weren't the actual object. The final shader was a sepia-tone for the out-of-sync memory zones that objects needed to be moved to. The effect would draw the player's attention to the area, and signify that this was a region that an object needed to move to.

Due to complications with the Source Engine, however, we were unable to implement these shaders within our game. Source engine's shader implementation was the only real hurdle we couldn't overcome on this project. The actual process of writing a shader is easy enough since it uses standard HLSL; the trouble with shaders in Source comes from the need to compile them and get them to work for your mod. Compiling shaders requires a very precise setup of your machine, getting specific programs, setting path variables, downloading old files, using an older operating system, and older Visual Studio/direct programs.

The problem is that even though we followed all these steps, shaders still wouldn't work. A lot of time was sunk into trying to get shaders working, mostly requiring us to slightly change things, recheck if everything was set up correctly, and just trying different things in the hope it would work. This problem was compounded by the fact that compiling shaders takes around 4-6 hours, and only 1 of our computers could actually compile them. We spent a fair amount of time trying to get this to work, communicating with people on the Source Development forums, where other people were having the same or similar problems compiling shaders.

In the end, shaders were simply not working, and we realized that they might never work. We abandoned them, and began using various other methods to achieve the same effects. The glow effect to show scanned items was used in a previous Source game, *Left 4 Dead,* and we had originally wanted to simply use the same effect. Despite having a tutorial on how to implement this feature, we couldn't do this due to missing texture files. Thankfully, shortly after we abandoned shaders the texture files were restored on the site, and we were able to implement the glow that is used in Obscura.

Since we couldn't have a see-through effect for projections we decided to use the glow, but a different color to represent it being a projection. We then used the same color and effect for associated objects, in effect making it easier to understand the similarities between the two mechanics. We did spend some time deciding on which colors to use to signify the effect. We went with a dark blue for scanned, a light blue for the focused item that is currently selected, and a yellow glow for projected/associated objects.

To replace the sepia of sync zones, we chose to use a lighting and halo effect to highlight the places where your memory is in need of repair. We had considered using color correction, so that when you entered a sync zone the entire screen would be sepia, but it proved to be too subtle an effect and wouldn't work in our mod anyway. For some reason, color correction requires you to place the files in the HL2 folder (as opposed to the folder your mod is in) so we would either have to check which folder they're using and make an executable place it there, or have them place it themselves. We deemed this too intrusive, and simply chose to work with lighting.

Developing the Game - Art

An Artist's Experience with the Source Engine

The Source Engine is, for artists not familiar with the engine, extremely finicky. The unfinished nature of most of the documentation for those that choose to create 3D assets using Autodesk Maya means that users could possibly run into problems at every step of the art pipeline. The art pipeline for those using Autodesk Maya is a 4-5 step process:

1. After the model is completely built, separate layers need to be built: a Reference layer and Physics layer
2. Once a collision mesh is created, combined, edges smoothed, and placed in the Physics layer, and a specific Phong material is applied to every model, an SMD export plug-in needs to be used.
3. The plug-in will spit out on average of 3 .smd files (each containing node positions along with the material source file to be applied) and a .qc file (commands to be run by compiler). The .qc file will be broken.
4. Once the .qc file has been configured properly, the model can finally be compiled by running the .qc file through Valve's compiler program called studiomdl. Compiling textures to the proper format involves using a similar program called Vtex.

Valve's developer community wiki site is sorely lacking in accurate instructions on how to get assets made, exported, and compiled, because of this something would go wrong at every point in the pipeline. This would cause problems ranging from broken/missing textures to completely broken models. Problems would arise for numerous reasons from the outdated nature of the Maya SMD export plug-in, and also because the articles would neglect to mention

small details which would cause the exporter and/or the compiler to fail. The kinds of details they neglect to mention include:

Assigning layer membership to the model itself as opposed to a pointer to the model that Maya by default assigns membership to

- The file name of the compiled textures has to be the same as the .tga used while texturing in the modeling software.
- Little details about the .qc files

On average, about as much time, if not more, was spent on fixing the broken models as was spent on making them. Given these problems, combined with the number of requested items and the length of time given to complete the project, the quality of the assets by the time the project deadline had arrived were (in the opinion of the sole artist) sub-par, almost all of which were given placeholder textures made by editing 4-5 base texture images in Photoshop. Around a total of 36-40 model assets were made from scratch, only about 10% of which were taken to a finished state.

On the other hand, despite the Source Engine's age, it's a perfectly capable engine. It's model polycount limit is somewhere around 10,000 vertexes per model, enough to have fairly high detail models made and re-topologized in high-poly sculpting programs such as ZBrush. Using GUIs for texture and model compiling also makes things significantly easier, especially in compiling textures. The Source Engine is capable of taking in texture files with different alpha channels and even animated textures. Models can even be outfitted with multiple skins.

Visuals

The proposed art style of the game has changed a lot since the original concepts and ideas we had. Originally, with our janitor idea, the plan was to make the whole place as dirty and rusty as possible. Once that idea was scrapped, we chose to start focusing on more modern or regular scenes. After we had finally settled with a story for our game, the plan was to make the whole place as gray and unclear as possible. We had around 7-8 levels, and each was going to be a mix of different environments. For example, one section of a room would be in a bedroom, and the other half would be an office with a cubicle in it. In the end, we figured that story-wise it would be a better idea if we told it like a series of memories leading up to the finale.

In terms of visual feel, we wanted the game to get darker and redder in color as the story progressed. While we did keep some aspect of this in the end, we decided not to make everything redder as well.

One of our biggest decisions we made involves the use of decals to describe the visual feel of the rooms you are in. Rather than populating the entire level with objects that could be used to break the game, or make the puzzles trivial, we decided to have all the unimportant models that wouldn't be used be displayed as flat decals projected onto a surface. This was done so that the player would be able to distinguish what items were important and what items were not, a problem we had been going over a lot during development. Not only would it let the player know what objects they have to work with, but it was also done so that it could potentially aid the dream-like and weird visuals.

Developing the Game – Sound

Sound Effects

We wanted each mechanic to have its own sound that made sense not only from what the character was doing, but what was physically going on in the world due to the character's action. When the character uses Transfer, he snaps his fingers. Associate, like Transfer, has a very physical element to it - the character claps his hands. For these two sounds, we decided to have the sound match the action of the character, as this action made its own sound.
For Project, however, the character flicks his hand and a new object appears. For this sound effect, therefore, we decided the sound would accentuate the action of the mechanic (creating a new object in thin air) as opposed to the action of the character (pointing). To this end, we chose the sound of a whip cracking, which we felt matched the hypothetical pop of air that would come from an object suddenly materializing.

Focus presented an interesting challenge. Unlike the other mechanics, it had no physical impact on the world - the player merely points, and the object he points at begins glowing. As such, there was no "natural" sound created by the player or mechanic. To this end, we decided to give Focus a very subtle wind sound, to give some aural feedback to the user about what they're doing.

All the sounds we used were from the free sound project at freesound.org. All the clips we used were released under a Creative Commons license, and the creators are credited in the game's credits.

Music

We used selections from Nine Inch Nail's *Ghost I-IV* as the soundtrack for the game. We felt that the mood behind many of the pieces, especially the ones we used, matched and accentuated the mood we wanted for each individual level, as well as the game as a whole. Like the sound effects, the album was released under a Creative Commons license, without which we would not have been able to legally use it.

Dialog

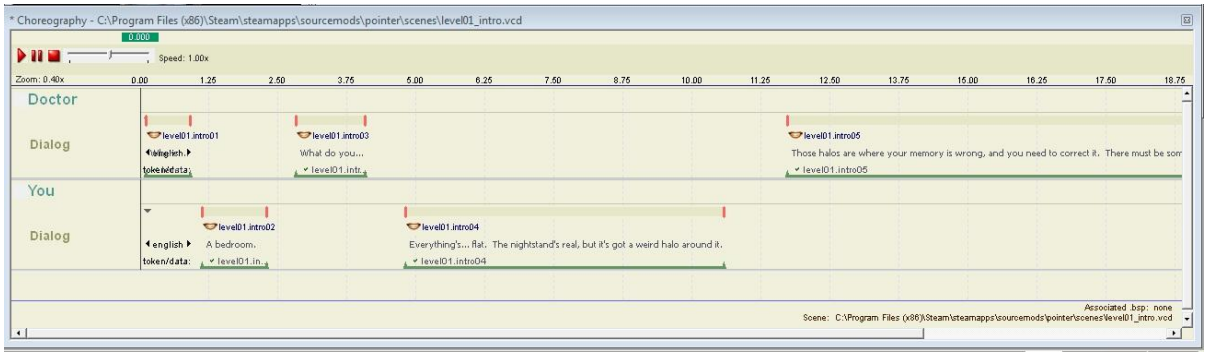
As the character development and story is mostly dialog-driven, and save the opening and closing scenes we only knew the characters by their dialog, we felt it important to find the right voices to portray the characters. While we sent out a casting call to the different theatre and comedy groups here at WPI, we weren't sure what kind of response we would get, based on the quick turnaround time we were looking for and the busy WPI end-of-term schedule.

Fortunately, we received enough replies that we needed to hold auditions. We met with each person interested and recorded them voicing a small number of lines. Each of us listened to these performances and we collectively chose actors for each role. The Doctor would be portrayed by Elena Ainley, and the Patient by Ben LaVerriere.

We were also fortunate in that we were able to work with the WPI Recording Club and utilize their facilities to record the final dialog for the game. We were able to get into contact with them through Elena, and after a good amount of schedule-searching we were able to find a time where the three of us, the two actors, and a technician from the Club, Robert Connick, could meet and record the dialog. Recording took a little over an hour; most of the lines were done in one take, with a couple of the lines done over a couple times for effect.

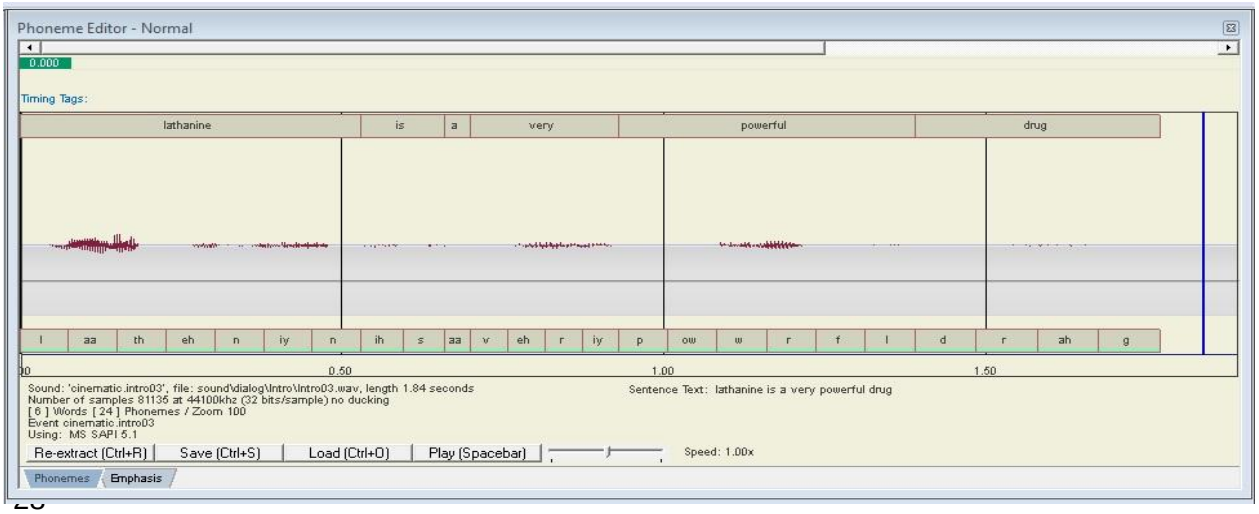
The way Source handles dialog is through scenes, which are created and edited through a program called Face Poser, which has an interface similar to film editor such as Final Cut Pro

or Adobe Premiere. For every scene within the game (except the opening and closing scene), we simply had to add each piece of dialog to the scene at the appropriate time.



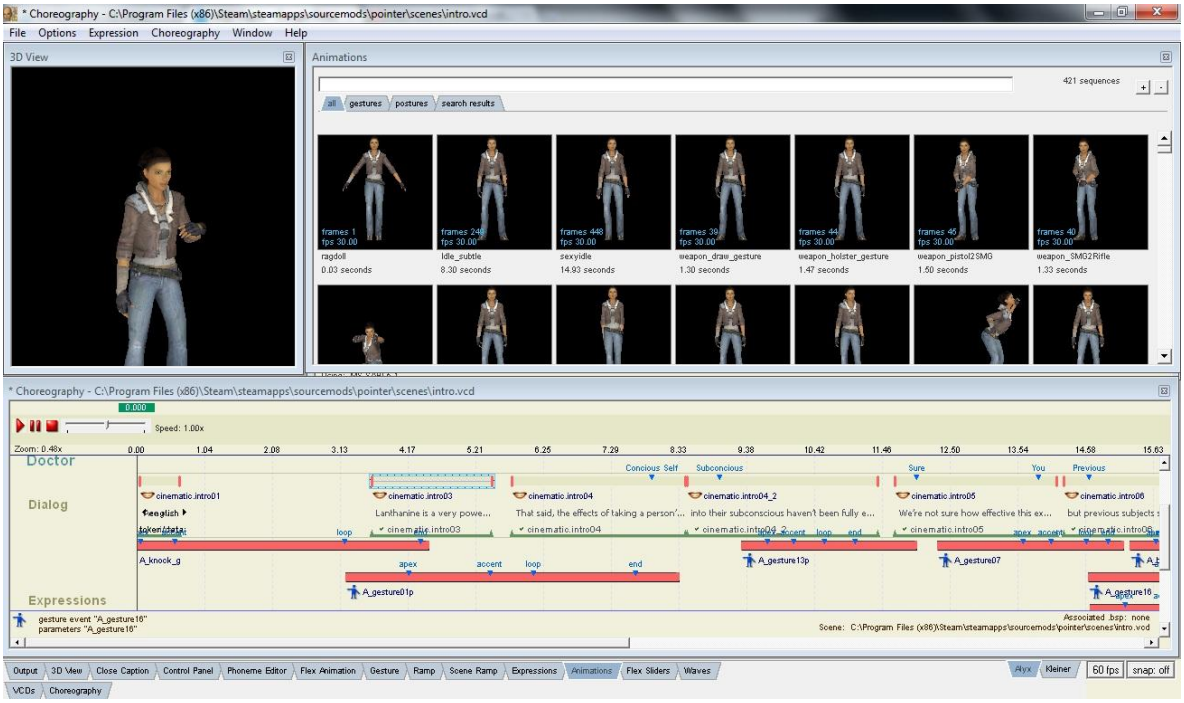
Editing a Scene, with the Doctor and the Player (You)

The opening and closing scenes were slightly more involved due to the fact that one of the characters - the Doctor - was on screen, and would need to speak and move accordingly. The first step in this process was adding phonemes to the individual .wav files. This also involved Face Poser. A .wav is loaded into the “Phoneme Editor” and the text said in the .wav is manually typed in. The editor is supposed to automatically calculate the phonemes on its own, but as we were doing this on a Windows 7 machine (which uses the Microsoft Speech API version 8, as opposed to version 5.1 which the system required) it would split the file into the separate phonemes, but would require someone to manually choose each one. While time consuming, it was definitely worth it in the end, as the model portraying the doctor convincingly spoke the dialog.



The Phoneme Editor Interface

For the movement of the actor, Face Poser has two separate systems: expressions and gestures. Expressions control the posture and facial expressions of the character, while gestures control the arms and legs. Expressions, for example, allow a character to stand up as if at attention, or slouch, while gestures allow the actor to point, wave or gesticulate with their hands. Gestures and expressions are added and handled like sound files, with a series of prebuilt expressions and gestures built in. The system also includes a model viewer, so you can watch the models in the scene as they move and speak.



Modifying a Scene with Actors, Gestures and Expressions

In the end, we were able to successfully use Face Poser and the Phoneme Editor to create a very convincing set of cutscenes to bookend the game with. While buggy, the mouth movements created by the Phoneme Editor, and the expressions and gestures put in with Face Poser literally and figuratively brought these two scenes to life, and made them both interesting and entertaining for the player.

Playtesting

Playtesting is always a humbling experience, and it's no different for Obscura. We had thought that our bugs and issues in the game numbered somewhere around 20, but after 3 playtesting sessions and multiple iterations of testing and fixing we had over 150 bugs. These issues ranged from minor things like a trigger not working, to game crashing errors, to much more subtle issues that could only be found after a specific chain of events.

During a playtesting session, we would record everything a player did or commented on. They would then play through the game from start to finish, and at the end we'd let them play around in the test room. Our guidance would be minimal at best, either explaining known issues, or giving them clues about model/textures that were currently not in game.

The major changes that occurred from playtesting, aside from bug fixes and making sure that everything worked exactly as it should, was giving information to the player. We realized that some things that were easy for us to understand were not explained to the player. Examples of this not needed include Associate's ability to transfer the weight of the focused object to the associated one, and the ability to undo a Transfer by unfocusing on the object that was transferred. We then added more dialogue, sounds, and HUD hints to inform the player about important gameplay elements.

Conclusion

We set out at the beginning of this project to create a fun and challenging game based around a series of mechanics we had thought up. As most of our work here at WPI had concentrated on development, we wanted to explicitly concentrate on the design of the game, to create the most cohesive and fun experience as possible. We feel that we have not only met our goal, but surpassed it.

That's not to say that the project didn't run into problem. We spent a lot of time on shaders that ultimately led us nowhere, and as the game only had one artist, we ran into a lot of timing issues in terms of what he would be realistically able to accomplish. But we recognized these problems and others, and found solutions for all of them, and carried on. Every effect that we were going to use a shader for, we found an alternate non-shader effect that worked just as well. And all of our art was carefully prioritized so that the things that absolutely needed to get done did get done.

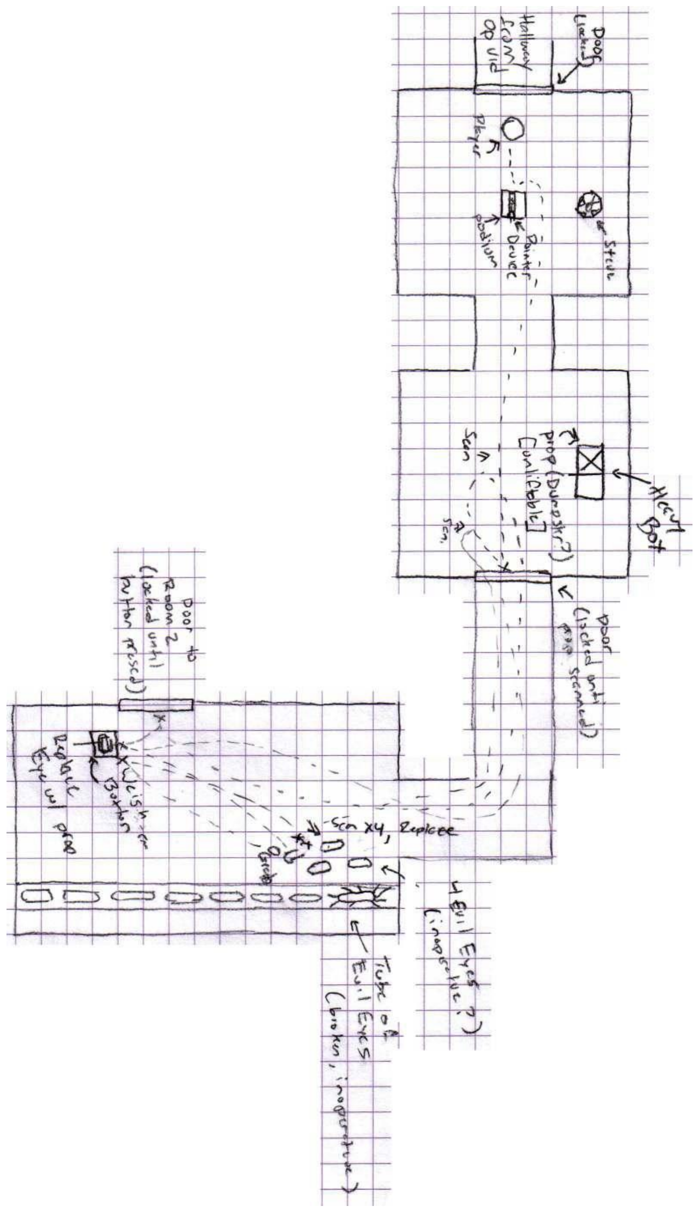
A game is never truly complete. Development never really ends. There is always something else to add, something else to make better. But at a certain point, the game is complete enough. The ship date is rapidly approaching, and the people who play it enjoy it immensely; the problems that seem obvious to the developers are missed entirely. It is at this point that development on the game stops, and the game ships. The game may not be complete, but it is done.

We are proud to present, to our advisor and to the students and faculty of Worcester Polytechnic Institute, our MQP: Obscura. It's not complete, but it is, finally, done.

Level Maps

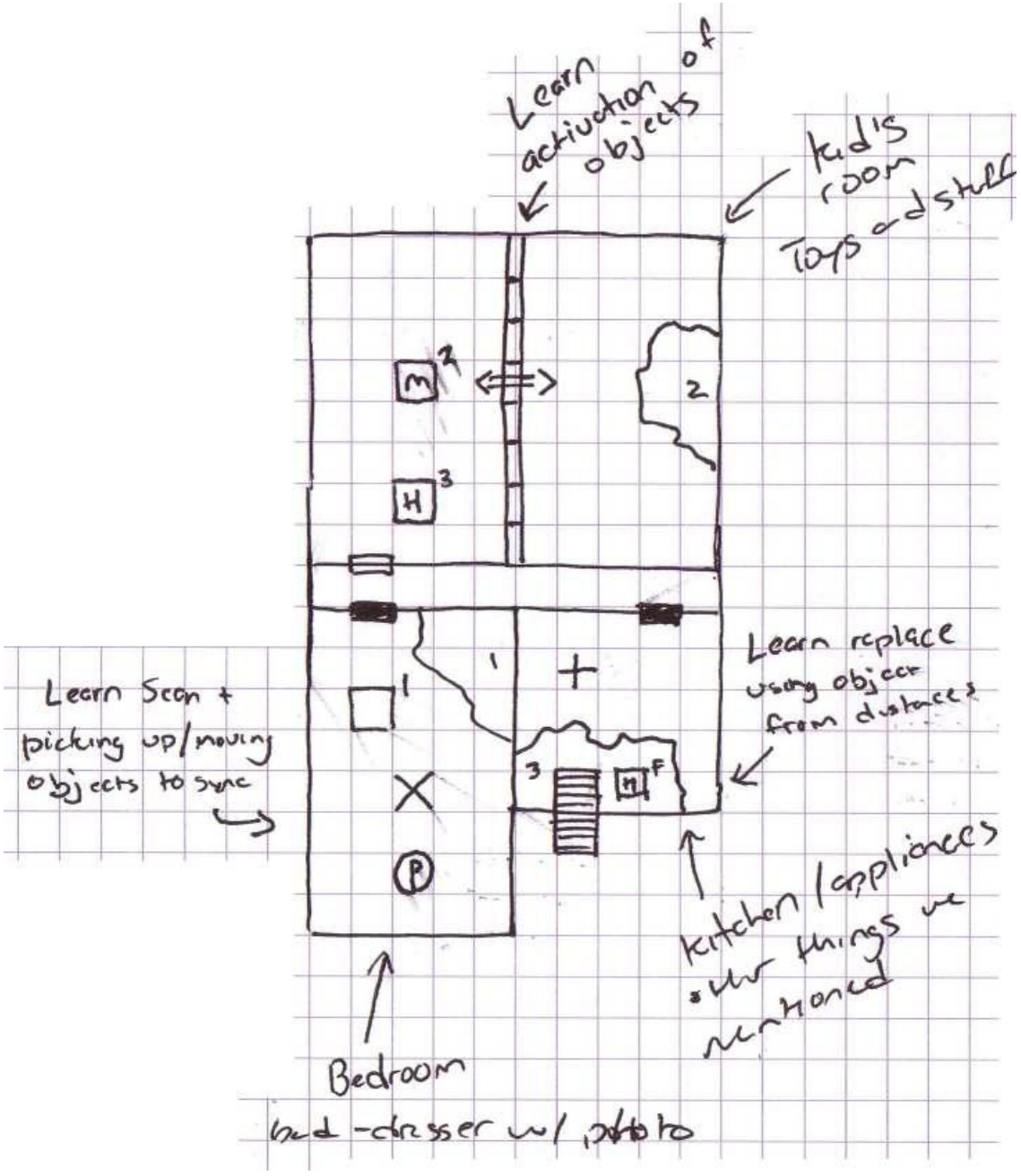
Original Level 1

This map was drawn out and planned while we were still working with the “Janitor” story. As such, it has a very *Portal*-like feel to it. In the end none of it was used; we rewrote Level 1 from scratch when we finalized the story.



Final Level 1

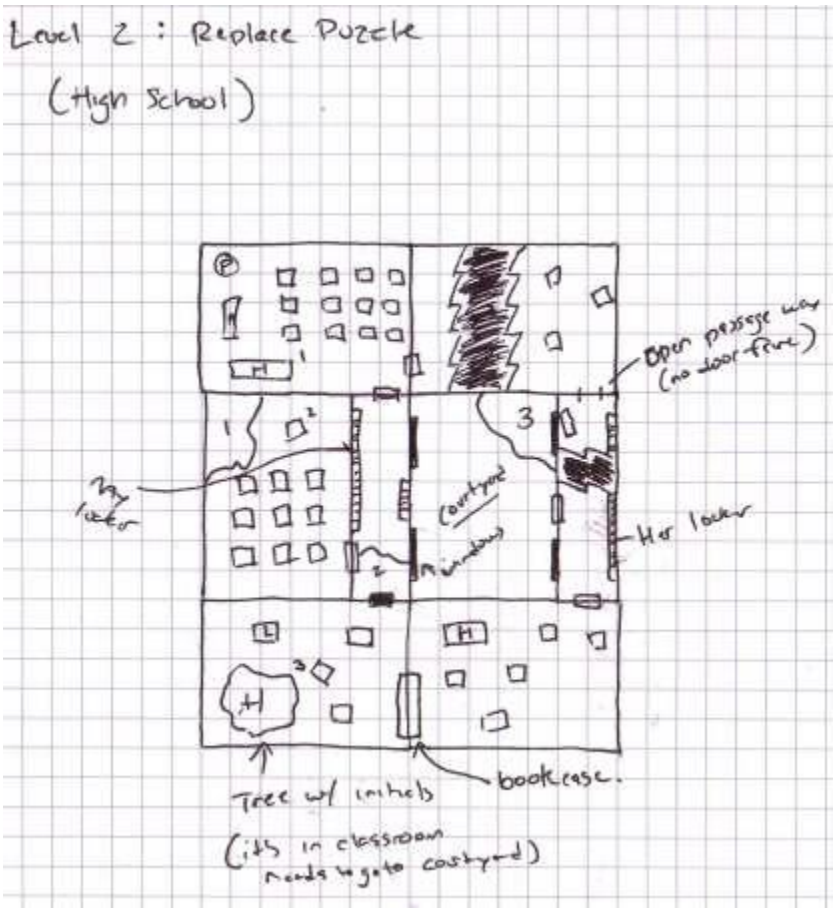
The final Level 1 is the player's home. The goal of the level is not to challenge them with any puzzles, rather to teach them the basics of the game - picking up items, activating items, focusing on items, and finally a very basic Transfer puzzle.



Level 2

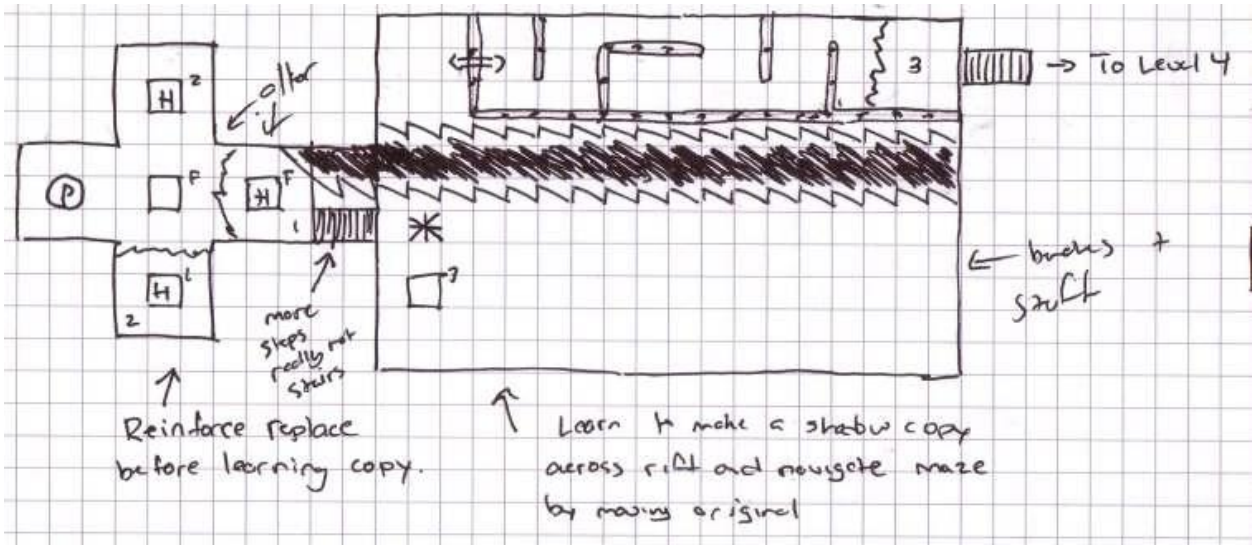
By the time we drew up Level 2, we knew what the story would be, and we knew that Level 2 would be the player's high school, where he first met his wife. As Level 1 was essentially a straight line, we wanted this level to feel more open, and with that in mind, by the end of the level it's a giant circle.

This level changed from its original map to the final product due to playtesting. Players found it odd to have a puzzle solution in a different room than the effect of solving the puzzle. To this end, we moved the 1 block (which in the game is a chair) in the left-hand center room to the same room you start in, and the 1 sync area to the top right room, which has the gap which closes when you solve puzzle 1.



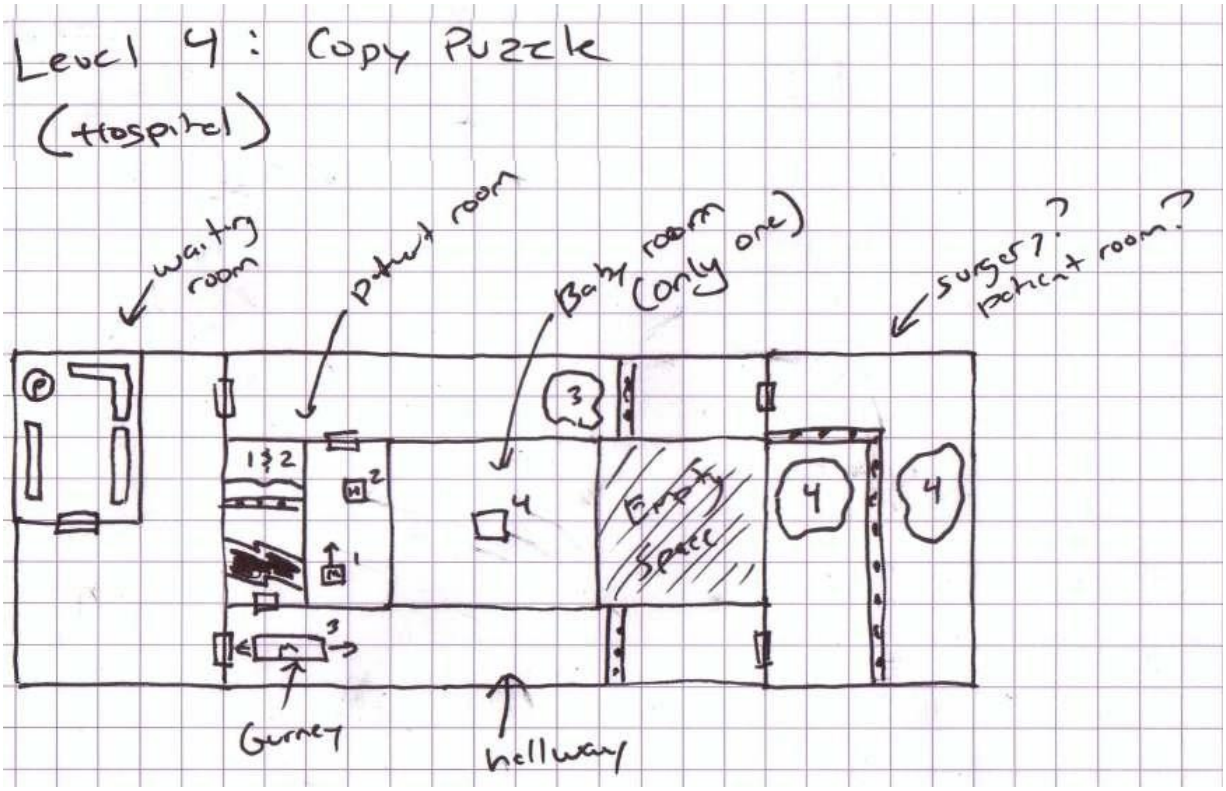
Level 3

The first goal for Level 3 was to reinforce Transfer one last time with the first area, and then to give the character the Project ability, with a simple puzzle that uses it. Along with Levels 1 and 5, this level was first mapped out during the “Janitor” story phase. Unlike those levels, however, the structure of this level never changed - we specifically chose to set this level in a church to keep the puzzles we created intact.



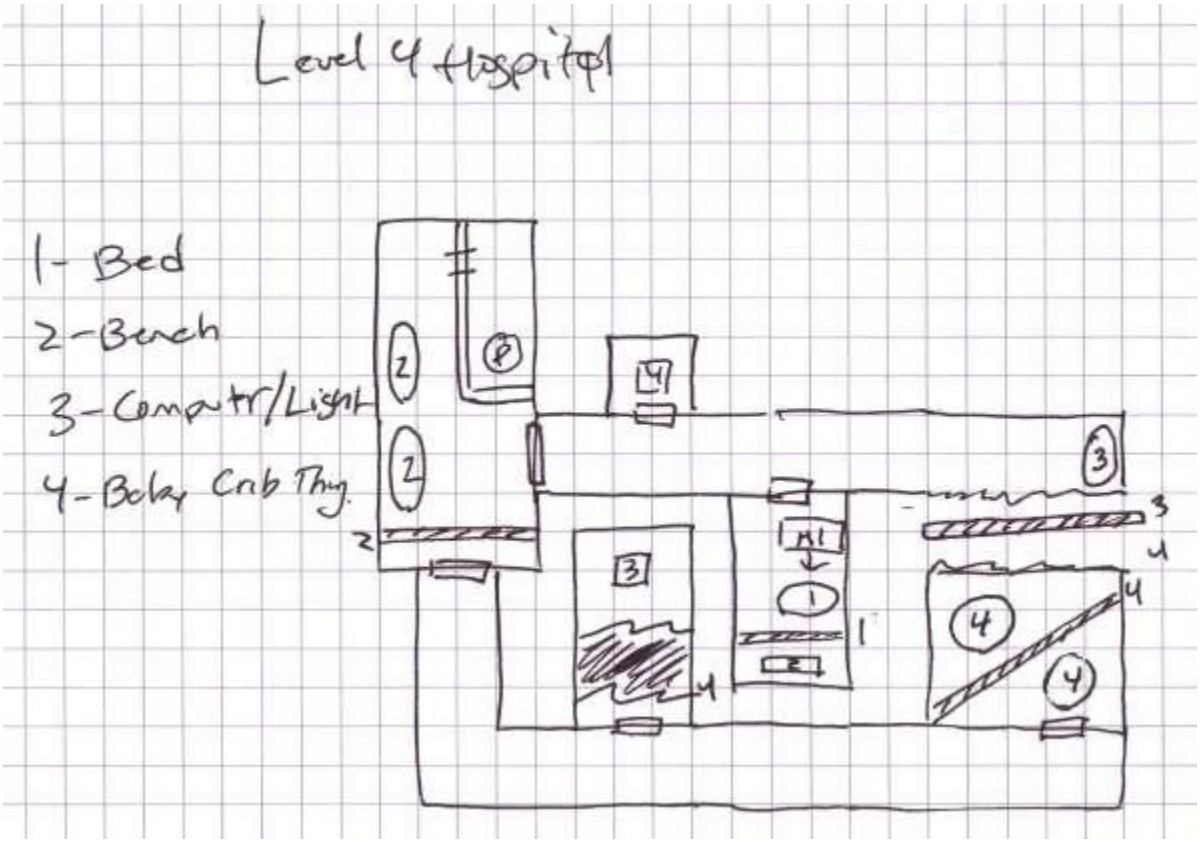
Original Level 4

The goal of Level 4 was to provide players with more challenging Project-based puzzles. The story takes place in the hospital where the character's son was born.



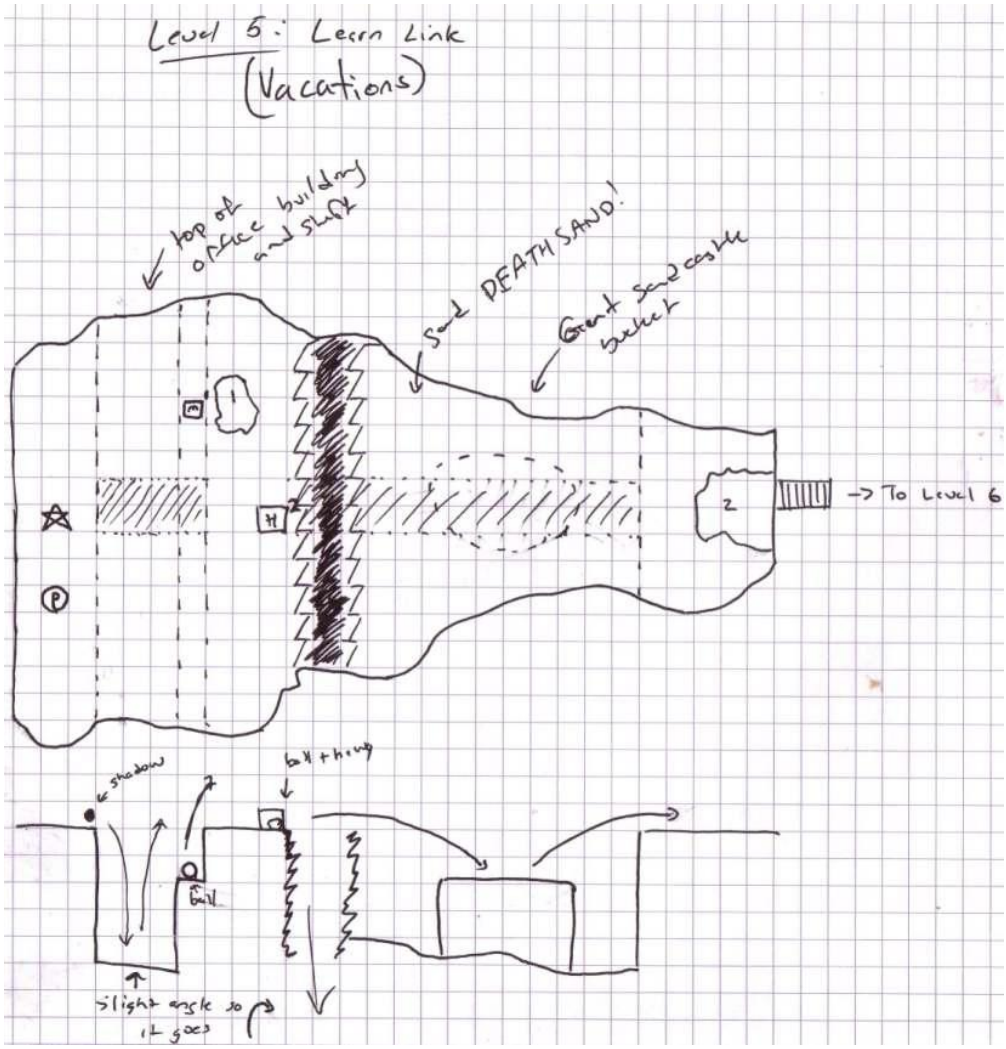
Final Level 4

Some minor changes were made to the level as it was developed; it opened up a little, and some of the puzzles were shifted around slightly to make sure that the game could never be put in a state where the player became stuck, or could no longer complete the puzzle without restarting.



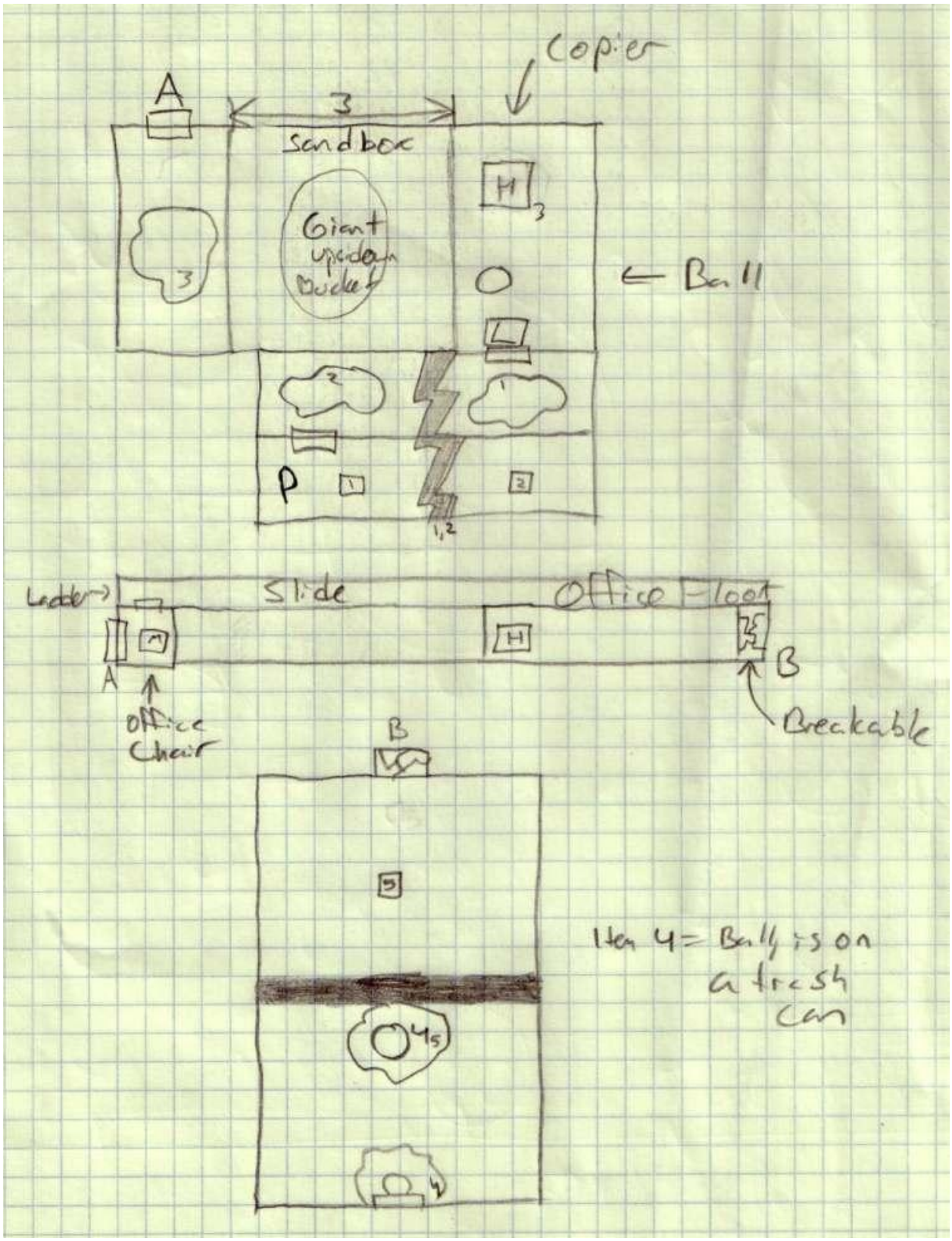
Original Level 5

Level 5 was first drawn up as an introduction to Associate at the very beginning of the design process, along with Levels 1 and 3. When we updated the story, the themes and details of this level were similarly updated as they were with Level 3. As people started playing the levels, however, they had no idea how to work through the level - it was too abstract and "geeky," and along with Level 6 the level was ultimately reworked from scratch.



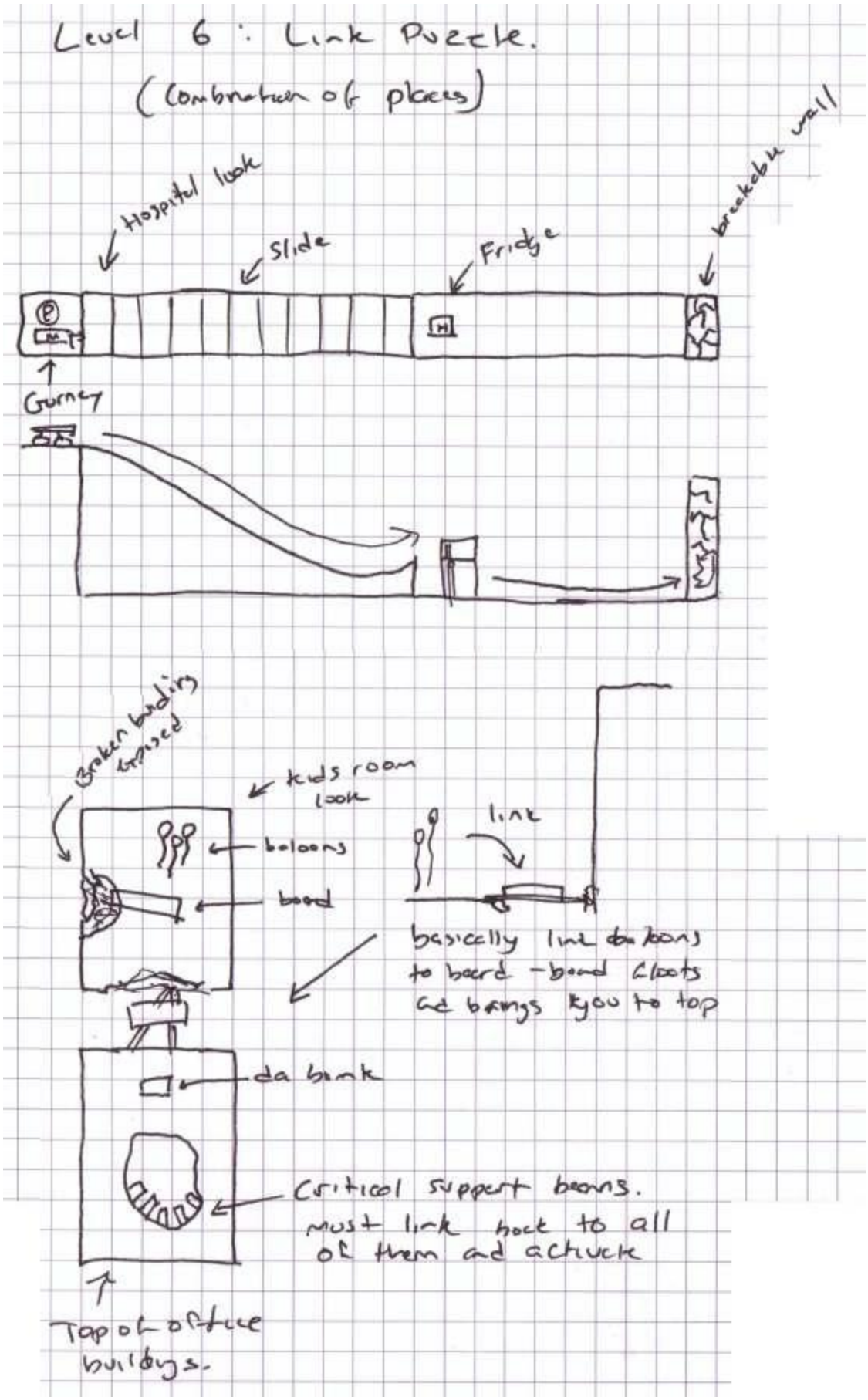
Final Level 5

The theme for Level 5 is “Office Play,” the idea being that parts of the level incorporate elements from an office and a playground. The slide puzzle was taken from the original design of Level 6 (discussed below), and in the final puzzle, the player must associate a whiskey bottle with a basketball, and then throw the ball through a hoop, causing the bottle to be thrown into the trash. The “reveal” at the end of this level is that the character was an alcoholic, and his family ultimately suffered for it. (Map on next page.)


Copier
A
3
sandbox
Giant
upsidedown
Bucket
← Ball
P
1,2
Ladder
Slide
Office Floor
A
B
Office
Chair
Breakable
B
Item 4 = Ball is on
a trash
can

Original Level 6

The goal of this level was to first provide the player with more difficult Associate puzzles, while at the same time show that the character is deep enough into his subconscious that there are no distinct memories; everything has become too disjointed and chaotic to get any single idea or memory. Ultimately, this level was dropped due to time constraints. The slide puzzle was incorporated into the final design of Level 5. (Map on next page.)

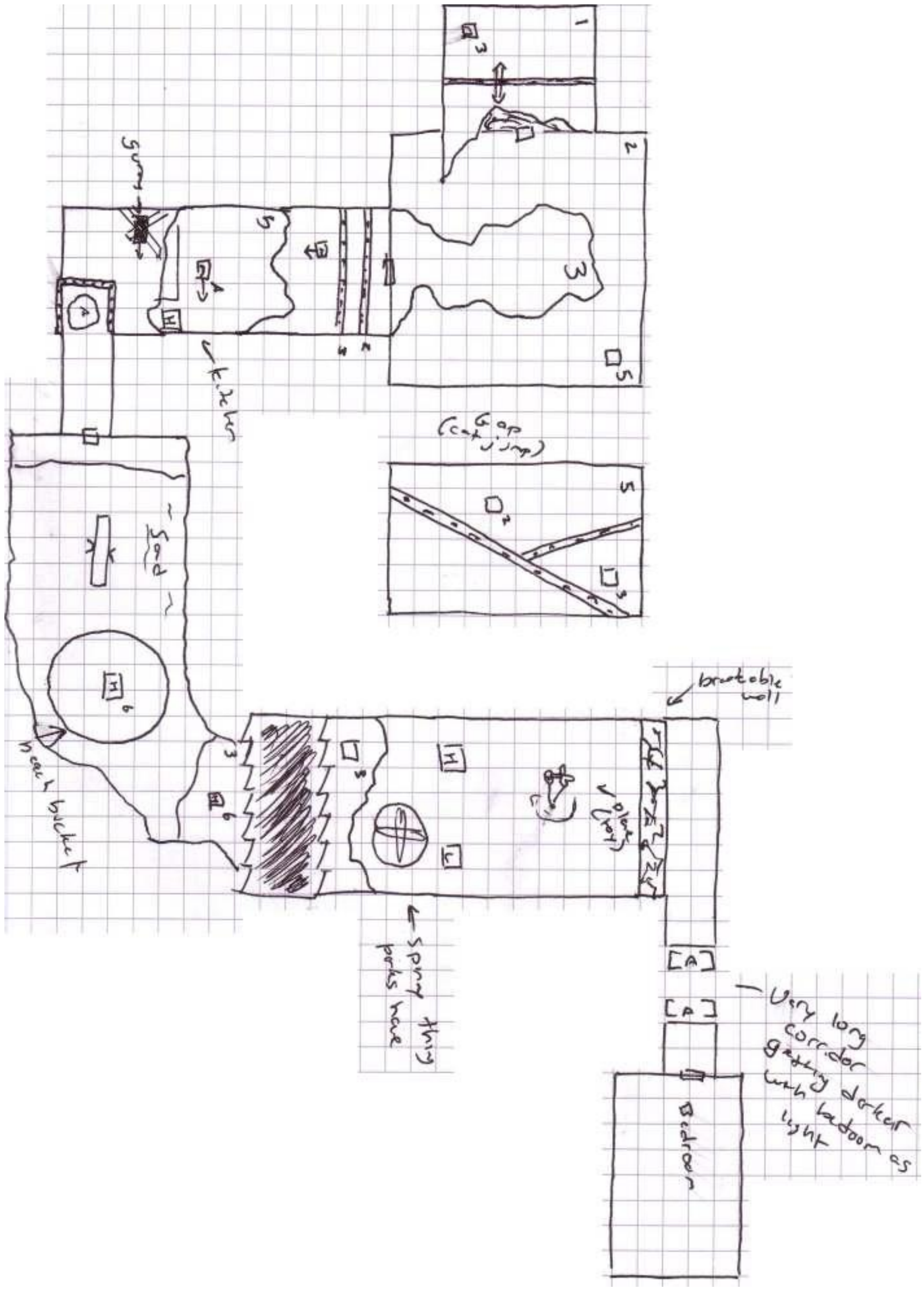

Level 6 : Link Puzzle.
(Combination of places)
Hospital look
Slide
Fridge
breakable wall
Gurney
Broken building exposed
kids room look
baloons
board
link
basically link da baloons to board - board floats & brings you to top
da bank
Critical support beams.
must link back to all of them and activate
Top of office buildings.

Original Level 7

The goal behind this level was to provide the player with hard puzzles incorporating all of the mechanics together. The numbers in different sections of this level represent the different regions of his memory that those areas are from.

1. Church
2. Building Top
3. School
4. Hospital
5. Home
6. Sandbox

This level was ultimately rewritten because it was too big for the time we had to do the project, and some of the puzzles, were too complex to be easily explainable. (Map on next page.)


kitchen
sand
near bucket
Gap (cat jump)
breakable wall
Very long corridor getting darker with bedroom as light
Bedroom

Final Level 6

The final design of this level is similar to the original design. The only major difference is that puzzles too complex to build or too trivial to solve have been dropped.