Abstract

The goal of this project was to design and develop a working demo for a two dimensional puzzle-platformer video game. To gain a better understanding of the processes and components necessary for game development, this project was undertaken without the aid of a game engine. Instead our game, currently titled Cosmo, was built from the ground up using C++ and the Simple DirectMedia Layer (SDL) development library. Cosmo was conceptualized and developed in collaboration with graphic designer David Shatan-Pardo ’15, who is responsible for the game’s aesthetics and visual design. The result of our collaboration is a two dimensional puzzle-platformer designed for Windows and Mac OS.

Motivation and Background

I have been playing video games for as long as I can remember, yet previous to the development of Cosmo, I had never undertaken any projects pertaining to game design and development. As a topic in which I had little background experience and a strong interest, I considered the development of a video game to be an ideal focus for this project. Additionally, while the Computer Science department at Yale offers many challenging and intriguing opportunities for its students, there seems to be an absence in its push for interdisciplinary projects. Having a strong interest in both technical and artistic design, I decided to work on a collaborative project that combines both.

We adopted the idea of developing a 2D puzzle-platformer for two separate reasons. First, without the use of a game engine, developing any substantial game in three dimensions is incredibly difficult and almost certainly outside of the scope of this project. Two dimensional design is significantly more accessible and provides a stable jumping off point for first time game developers. Additionally our decision to create a 2D puzzle-platformer stems from our own gaming preference. Both my colleague David and I have
a strong interest in adventure and puzzle games, and some of our favorites have used the two dimensional puzzle-platformer concept. Many of those games can be seen as inspiration for this project, including games such as Portal, Fez, Braid and Super Brothers Sword and Sworcery EP.

The decision to avoid the use of a game engine stemmed from a desire to delve into the inner mechanisms of game design. Game engines make development faster and simpler, as they’re intended to do, but eliminate the need to implement various aspects of gaming such as collision detection, physics, rendering and load management. These are all components that are essential to video game development, and thus are all extremely important in understanding how a game functions. Developing without an engine provides insight into these components, and while this type of development may not yield the most optimized results, it is an incredibly valuable learning tool.

**Gameplay**

As stated above, Cosmo is a 2D puzzle-platformer. Similar to games such as Super Mario Brothers and Super Metroid, Cosmo focuses on moving an avatar through a sequence of two dimensional levels while relying on both the skill and puzzle-solving abilities of the player. The player’s avatar is able to move laterally through levels by walking or running, as well as vertically by jumping, a defining characteristic of the platforming genre. Like many platformers, the player’s avatar “dies” when he falls below the vertical bounds of the level. In our development of Cosmo, we placed a higher value on the player’s desire for exploration and puzzle-solving than his or her in-game motor skills. Cosmo was not designed as a skill based game, but rather as a game that rewards curiosity and creativity.

Given that 2D platformers represent a rather common genre in the world of video games, it is important that a game has some unique mechanic to set it apart. For Cosmo, this mechanic aligns with our emphasis on exploration. Each level designed for Cosmo consists of three separate worlds. Each world has its own aesthetic, determined by both color and environment, with the first being a blue cave world, the second being a red forest world and the third being a yellow world focused on ancient architecture. Within a level, the layout of each of these worlds is incredibly similar, yet each has subtle differences that give the player the ability progress through the level, solve puzzles, and access different areas of the game. An example of the differences and similarities between these worlds can be seen in Figure 1. At any given time, the player’s avatar occupies physical space in only one of these three worlds, however by pushing a button or stepping through a teleporter, the player is able to move their avatar between these parallel worlds.
Figure 1: Examples of the blue, red and yellow worlds. Notice how the platforms in all three worlds are identical, each expressing a three-tiered structure, with the highest platform on the right. Subtle changes exist between the worlds, such as the positioning of doors and ladders.

As the narrative of the game is currently a work in progress, the overall objective of the game is still rather ambiguous. The basic premise however, revolves around progressing through levels while collecting a variety of objects, with some objects serving as optional extras or Easter eggs, and some serving as essential to the plot. A certain number of required objects will be needed for the successful completion of the game. A simple example of this would be collecting parts for a broken rocket ship, a basic implementation of which can be seen in our demo. As the narrative and background of our character develops, the overall objective with become more definitive. Regardless, the game will continue to apply the same focus on exploration and adventure.

Design

Before discussing the in depth implementation of the Cosmo, it is important to understand the organization of the game at a higher level, and the basic components used that provide the game with functionality. As mentioned previously, Cosmo is a level based game, with each level containing three similar, yet distinct worlds. As can be seen in Figure 2, each world is comprised of a variety of different components. Understanding each of these components is essential to understanding the underlying organization of Cosmo. Here is a brief overview of each component:

Environment Textures: All visual effects for Cosmo are accomplished through the rendering of textures. Each world has a specific main texture which is composed of the world’s basic landscape and static platforms. Additionally, each world may contain an optional foreground and background, which experience parallax effects as the player moves his or her avatar throughout the world.
**Tileset:** The tileset is used to store the underlying grid system of *Cosmo.* Like most platformers, our game uses a simple grid system for static collision detection. Most static elements, such as basic platforms, slopes and ladders are stored in this grid of tiles, referred to in this paper as a *tileset.* Each tile in our tileset is has 16 x 16 pixel dimensions, and each tile has its own tile-type, signaling how the avatar interacts with that specific cell of the grid.

**Platform Set:** The platform set is used for storing dynamic platforms. Since moving platforms do not strictly align to a grid system, they cannot be stored in our tileset. Platforms may move at varying speeds in either the horizontal or vertical direction. Because these platforms do not adhere to a grid system, they adopt a different approach to collision detection, which will be discussed later.

**Objects:** Objects represent anything the player may be able to interact with during their gameplay, excluding doors. These objects may include: teleporters, keys, buttons and various artifacts the player may find as they make their way through the game. As objects may be transported between worlds and levels, objects are not stored on a per world basis, but are instead stored globally. Each world refers to the global list of objects to know what objects currently exist in that world.

**Doors:** The same idea applies to doors. Doors are the basic mechanic for traveling between levels. As any given door requires knowledge of the existence of doors in other worlds and other levels, doors are stored in the same global style as objects. Again, each world refers to the global list of doors to know which doors it must render, and the state of each door - primarily locked or open.

![Figure 2](image_url): The basic component of which a world is comprised.
Implementation

Collision Detection
Collision detection is perhaps the most important aspect of any video game. It provides the basis for an avatar to stand on a platform, run into walls, and climb up slopes. Without collision detection, a player would be free to move their avatar anywhere within the xy-plane of a world. Collision detection is used to provide landscape, boundaries and interaction with the surrounding world.

Due to the differences in our representation of static and dynamic platforms, we use a few different variations of collision detection within Cosmo. There are two major methods for static collision detection, basic detection and slope detection, as well as one uniform method for dynamic detection. Before analyzing collision detection however, it is important to comment on the basic movement style of our in-game character. Our two dimensional movement is divided into horizontal and vertical components, both with a direction and a magnitude. Horizontal movement and collisions are processed before vertical collisions and the surrounding world.

Basic Static Detection: Basic static detection works only with our tileset, as it relies heavily on the grid system. Given a velocity, as well as the avatar’s current location, we can determine which cells in our tileset may potentially cause collisions, as can be seen in Figure 3. By first analyzing the cells closest to our avatar, and then moving outward, we can ensure that the first collision we detect will be our most immediate obstacle. We then determine the distance between our avatar and this obstacle, and update our position accordingly. If no collisions are detected in the selected group of cells, the avatar’s position is adjusted by the full value of the velocity.

Figure 3: Basic static collision detection. From left to right, each cell contained in the red portion is checked for static tiles. The avatar is moved adjacent to the first obstacle encountered.
Slope Detection: Slope detection uses similar logic as our basic collision detection, however, when approaching a slope from the front, there is no horizontal collision. Instead, the avatar moves into an overlapping position with the slope. This overlap is then remedied in via vertical collision detection. After detecting an overlap with our avatar and a slope, we calculate the vertical distance needed, given the avatar’s horizontal positioning, to reach the edge of the slope. This process involves analyzing which pixel of the slope the leading edge of the avatar currently overlaps with, as well as the steepness of current slope. After the analysis, the avatar’s vertical position is adjusted accordingly.

![Figure 4: Slope collision detection. Slopes do not serve as horizontal collisions. Rather the climbing effect is achieved my adjusting the vertical position when an avatar is overlapping a slope.](image)

Dynamic Detection: Dynamic detection is used for our platform set, as well as objects. Despite the fact that objects may still be static, they are not required to align to the grid system, and thus must use a different form of detection. Dynamic detection is simpler than the previous two methods. The first step is to update the avatar’s collision box with the new velocity. We then check to see if this new collision box overlaps any of the platforms or fixed objects in our given world. If so, we find the obstacle closest to our avatar’s current position, and update accordingly. If not, the avatar’s position is updated based on the full velocity. An example of this type of detection can be found in Figure 5.

As far as components that the avatar can pass through, such as doors, ladders, and non-fixed objects like keys, collision detection is irrelevant. However, it is necessary to know whether the character currently overlaps with any of these features. This helps to trigger events such as passing through a door or climbing up a ladder. For this sort of detection can be achieved by simply testing for overlap between the avatar’s collision box and the collision box of the object. This basic overlap test adds important functionality to our worlds.
**Figure 5:** Dynamic collision detection. The red portion represents the projected avatar position. Because the position overlaps with a platform, the avatar moves a shorter distance.

**Rendering**

Basic rendering for Cosmo was made easier with the use of the SDL development library. SDL’s texture rendering API provides an intuitive library for fast hardware based rendering. For Cosmo, all character, object and loading sprites, as well as world textures are stored locally as PNG files. Cosmo loads these files as necessary and converts them to SDL textures. These textures can then be rendered quickly and efficiently, giving the game its unique aesthetic feel.

In almost all two dimensional games, character and object animations are handled with the use of a sprite sheet. A sprite sheet is a single image that contains multiple frames of an animation. During gameplay, these sheets are clipped such that only one frame is displayed at a given moment. Using a single image for an animation allows for less overhead and faster loading times, and this technique is employed heavily throughout Cosmo.

**Figure 6:** Our original sprite sheet for the walking animation for our protagonist, Cosmo.
While sprite sheets are fairly standard for animation, there are multiple techniques employed by 2D games for rendering environments. Games that rely purely on a grid system often render each tile individually. This effect can be seen in games such as the original *Pokemon*, where each tile or group of tiles is represented by its own sprite. This is not the rendering solution used in *Cosmo* however. Given the size of each of our tiles, 16 x 16 pixels, and an average monitor resolution, say 1388 x 728, we can expect to be individually rendering over 4000 tiles every frame. We tested this strategy, and found a significant lag in our gameplay speed. Due to this, we opted for a more unique rendering strategy.

Instead of rendering each individual tile, we decided to store our entire static environment in three separate images, a background, a main texture, and a foreground, as can be seen in Figure 7. The environment represented in the main texture corresponds to the platforms stored in tileset for that specific world. This is true for doors depicted in the main texture as well. However, storing the texture independent of the tileset mean that any changes made to the tileset will not be automatically reflected in the environment. Due to this separation, our rendering system results in a decrease in modularity, but an increase in runtime efficiency.

![Figure 7: The background, main, and foreground texture layers for a basic environment in our red forest world.](image)

Characters, objects, locked doors, and dynamic platforms are rendered independently of the major world textures. For each frame, the rendering order is as follows: background texture, main texture, objects, locked doors, dynamic platforms, characters, foreground texture. Having the foreground texture as the final rendering allows the character to pass in front of various objects and doors, but to pass behind elements in the foreground of the game. Allowing the avatar to move behind some aspects of the environment creates a simple, two-dimensional, depth effect.

**The Storage System**

Currently almost all game data is stored in a hierarchical system of text files. Each component, such as an object list, a tileset or a platform set has its own method of
storage. This method is processed by our game when the specific component is loaded. Storing these components in text files allows for a quick adjustment of gameplay elements through the simple text editing. This is an ideal system for testing, although it may need to be adjusted if a completed version of the game is released publically, as it may be simple to hack. Simple encryption or compression should help resolve the issue.

**Loading**

Given the amount of overhead for a single world, and the fact that each level has three worlds, it is implausible to attempt to load all levels into memory at startup. Instead, a system is in place for loading and freeing levels as the player progresses through the game. As *Cosmo* is a non-linear game, we can always expect the player to return to a previous world, even if it has been freed. The loading system we developed is designed to keep as many neighboring levels in memory as possible, such that if a player is repeatedly moving between adjacent rooms, load time is minimized. A basic depiction of neighboring levels can be seen in Figure 8.

![Figure 8: A basic example of a level map. Movement through levels is handled by doors. Any levels immediately connected with an arrow are neighbors, and arrow colors represent in which world the door resided.](image)

During gameplay, no more than five levels are stored in memory at a time. However this number is variable, and may expand as more levels are developed. Entering a door triggers a loading sequence for a new level. If not already loaded, the destination level is added to our list of stored levels. We then find all the neighbors of the destination level, by looking the doors contained within its three worlds, and finding their connections. Checking this list of neighbors against our stored levels, we are able to find if there are any levels remaining in storage that are not adjacent to our destination level. If such a level exists, we free it from memory. If more than one exists, we free the first in our list.

The technique keeps our memory manageable, and as we are never loading or unloading more than one level at a time, results in generally uniform loading time. Loading of levels
is done in a thread that is separate from our general game thread. This allows us to render a simple loading screen while still loading and freeing levels.

**Level Design**

Beyond basic implementation, the value of a game can be determined by how well its levels are designed. Level design takes an entirely different approach from game implementation, as it focuses on a more creative and less practical approach. To aid in level design for *Cosmo*, we developed a simple tool for level building. This tool allows the user to select an image, generally the main texture of a world, and render this image beneath a grid representing a tileset. The user is then able to fill in sections of this tileset with specific tile-types, such as static blocks, slopes, and ladders. The initial image serves as a guideline for the user, as can be seen in Figure 9. The user generated tileset can then be exported to a text file, which can be added to our storage system and implemented as a playable level.

![Figure 9: A base image and an example of our level designer. The darker sections show static tiles, while the arrows represent pass-through platforms. Empty sections represent areas of the tileset without any obstacles.](image)

Given that the design of levels and the visual design were carried out at differing paces, it was important to establish a system for designing levels that did not rely heavily on aesthetics. Using Adobe Photoshop, a grid system, and a simple color scheme, I was able to design simple worlds independent of my designer David. These worlds were then input into the level designer and converted into playable levels. After creating these simple worlds, I would pass them on to David for enhanced aesthetic design. Switching between my basic design and his advanced visuals was seamless, as our rendering system relies solely on an input image. Many of the worlds contained in our demo possess my simple design, which will be improved by David in the near future.
Despite the talk of level design, we have failed to mention any method for the input of doors, objects, or dynamic platforms into our levels. This is a functionality that our current level designer is lacking. As of now, these components must be added manually, with the modification of the corresponding text files. Moving forward, this is something we hope to adjust in the near future.

**Future Work**

Our goal, in the upcoming years, is to turn this demo of *Cosmo*, into a full-fledged game. We have a long way to go, especially in terms of level and puzzle design, but we believe, for the most part, that the basic system is in place. There are few specific areas in which we would like to see improvements, such as a more robust level designer, a larger variety of intractable objects, additions to the user interface, and, of course, a stronger narrative and a longer game.

*Level Designer:* As discussed above, the current level designer can only generate tilesets. While these are important, as our levels become more and more complex, with move more moving parts, it will be essential to have a level designer that can handle the dynamic components of *Cosmo*. This will be an interesting undertaking, as beyond simply filling in tiles, there will need to be settings for controlling speed and directions of platforms, interactions of objects, and connections between doors. We additionally hope that while designing levels, there will be an option to switch into gameplay mode, such that the levels may be tested on the spot.

*Objects:* Currently our object repertoire is limited to keys, teleporters, artifacts and buttons. Keys are used for unlocking doors and teleporters are used for switching between worlds. The recent addition of buttons has added a huge increase in the potential for complex level design, as buttons can now make objects appear and disappear, unlock doors, and cause platforms to move. In the future, we hope to develop a more robust system for handling button interactions. Additionally, we would like to add a few more objects into the game, such as lore, which found by the player, will display some words providing history or mystery about the environment and worlds you are immersed in.

*User Interface:* We are also interested in improving the user interface to include a level map and an inventory of collected objects. As the game grows, these factors will be important for reminding the user of their current location and progress. Additionally, while we have built a rudimentary saving function, we have currently removed it due to the length of the demo. We would like to reinstate this function as well, to allow the user to experience the game over an extended period of time.
Narrative and Additional Levels: While many of the pieces are in place now, there are still a few more aspects we hope to achieve to push our game to the next level. Once we are satisfied with our basic and advance components, we will then truly begin to flesh out the narrative of our game and the background of our character. From there, we will start to build more advanced levels to provide the user with a fun, unique, and challenging game experience.

Acknowledgements

I would like to especially thank David Shatan-Pardo, my partner in crime on this project, as well as our advisers, Julie Dorsey and Johannes DeYoung, who have given us a flurry of good advice and interesting ideas. I’d also like to thank all of those who put up with my constant excitement and eagerness to show off this project, as well as everyone who volunteered to play, test, and critique it. Thanks especially to the Yale Computer Science depart for being open to fun and creative projects, and allowing me to pursue and educate myself on a topic that I truly enjoyed.