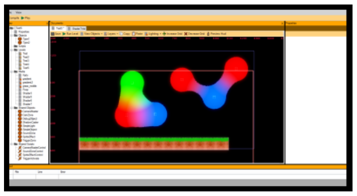
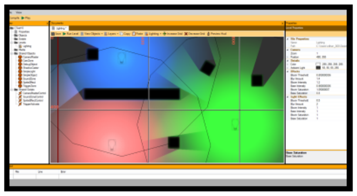
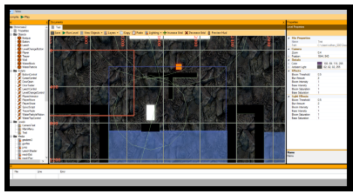
Worcester Polytechnic Institute

Whiskey2D
A 2D Game Creator

A Major Qualifying Project

Christopher Hanna, Daniel True
March 27th, 2015

Advisor: Professor Charles Rich

Table of Contents

Chapter 1 - Introduction

In the United States, the video game industry saw a 13.1 billion dollar revenue for 2014 (Statista 2015). The industry has been growing since the early 1980's and shows no sign of stopping. Thousands of games are released every year, mostly by independent developers. In order to release this many games, many of them share a common set of tools with one another, called a game engine. This engine facilitates rapid development for each game, often providing rendering, physics systems, scripting, and more. Generally speaking, engines are also bundled with customized Integrated Development Environments (IDEs), or game editors that can create games built for the engine. These game creation editors are invaluable for developers and supporting the large revenues seen by the industry. For this Major Qualifying Project (MQP), we made a 2D game engine and editor called Whiskey2D. The engine and editor, together, allow users to create and play their own 2D games from start to finish.

1.1 Project Goals

We had two main goals for the development of Whiskey2D. Firstly, we wanted our finished product to be fully featured, usable, and accessible to our target audience of hobbyist C# developers. Secondly, we wanted our development process to be well organized, efficient, and highly productive. Our mission was to act like professional developers, building robust software.

We wanted to build an engine and have it support the game creation process from beginning to end. That meant that we needed a core game engine, as well as a game editor. We also needed to create documentation for both components. We wanted our software to be complete and only have tested and usable features.

In order to create the engine we wanted to build, we needed to ensure a good software engineering process. From the beginning, we set out to do rapid development, highly iterative, and well organized software design.

Despite trying to bring a rich and fully featured game engine to life, we were not trying to create something completely new. There are already several excellent 2D game creation engines available for free. We were trying to build something comparable to the tools of the trade, build it well, and learn a lot along the way.

1.2 Target Audience

We wanted to keep Whiskey2D open and usable for as many people in our target audience as possible. We kept the target audience in mind by conducting several usability tests throughout Whiskey2D's development.

Throughout this report, the term *designer* will be used for the person who will be using Whiskey2D to build their own games. Designers will be expected to have a basic knowledge of object-oriented design so that they can understand Whiskey2D's constructs and services. Designers will also need to be able to write code in basic C#. The designers will not need extensive knowledge about how the editor works internally. They will only need a high-level knowledge about what services Whiskey2D exposes publically. We expect that the designers will be beginner to intermediate level game programmers. Whiskey2D will be built so that is easy to use for newcomers to game programming. However, it will also be just advanced enough to entertain more veteran game programmers. We expect that most of the designers will be hobbyist game programmers.

To avoid confusion, the designer is the person who creates games using Whiskey2D, while the *player* is the person who is actually playing the final game. Obviously, the designer and the player may be the same person, but the terms will not be used interchangeably. The player does not need to know anything about programming, C#, or Whiskey2D at all.

1.3 Related Work

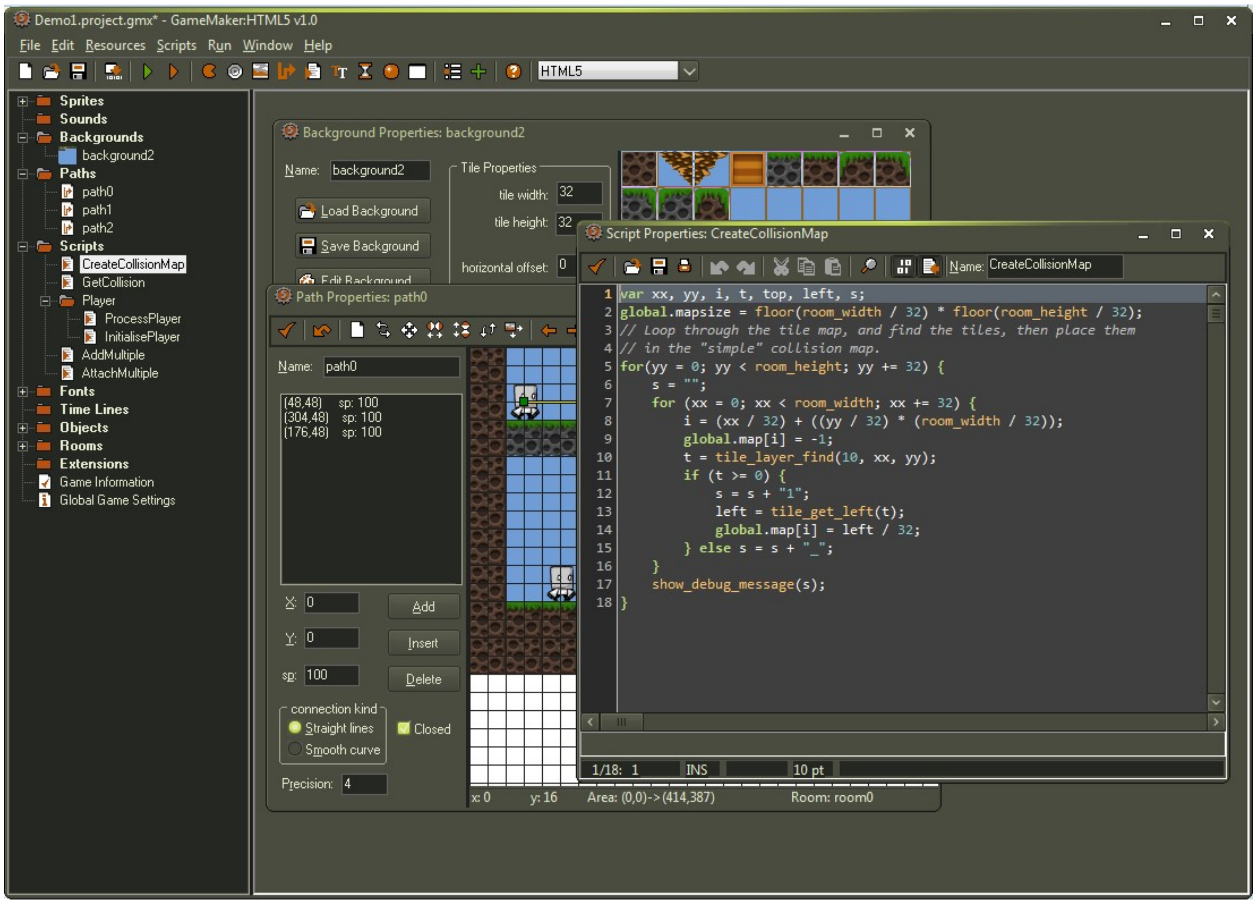
Before we began designing our system, we looked at a few examples of other game engines and toolkits. The three main examples we looked at were GameMaker, Unity, and Construct 2. Each provides its users with the ability to visually build a game, and deploy it to multiple target platforms such as Windows, Mac, or Linux.

1.3.1 Related Work Summaries

GameMaker

GameMaker[1] is made by Yoyo Games and is a 2D game editor and engine targeted at hobbyist game creators. The software allows users to load in or build sprites that can be applied to objects. The objects can be placed inside rooms, which is a GameMaker abstraction for a level. Each object can be scripted with a proprietary language called Game Maker Language (GML). The system is event driven, with some standard events included out of the box, such as initialization, updating, input, and destruction events. Each game has global properties, such as gravity, that can be accessed from any GML script. GameMaker has built-in editors for sprite creation, script editing, and map editing, which helps unify the game creation process. GameMaker can output games to several platforms. A screenshot of GameMaker running can be seen in Figure 1.1 below.

[1] More information on GameMaker can be found at https://www.yoyogames.com/studio



Name: background2
tile width: 32
tile height: 32
horizontal offset: 0

Name: path0

```
(48,48)    sp: 100
(304,48)   sp: 100
(176,48)   sp: 100
```

X: 0
Y: 0
sp: 100
Precision: 4

x: 0   y: 16   Area: (0,0)->(414,387)   Room: room0

Name: CreateCollisionMap

```
var xx, yy, i, t, top, left, s;
global.mapsize = floor(room_width / 32) * floor(room_height / 32);
// Loop through the tile map, and find the tiles, then place them
// in the "simple" collision map.
for(yy = 0; yy < room_height; yy += 32) {
    s = "";
    for (xx = 0; xx < room_width; xx += 32) {
        i = (xx / 32) + ((yy / 32) * (room_width / 32));
        global.map[i] = -1;
        t = tile_layer_find(10, xx, yy);
        if (t >= 0) {
            s = s + "1";
            left = tile_get_left(t);
            global.map[i] = left / 32;
        } else s = s + "_";
    }
    show_debug_message(s);
}
```

1/18: 1   INS   10 pt

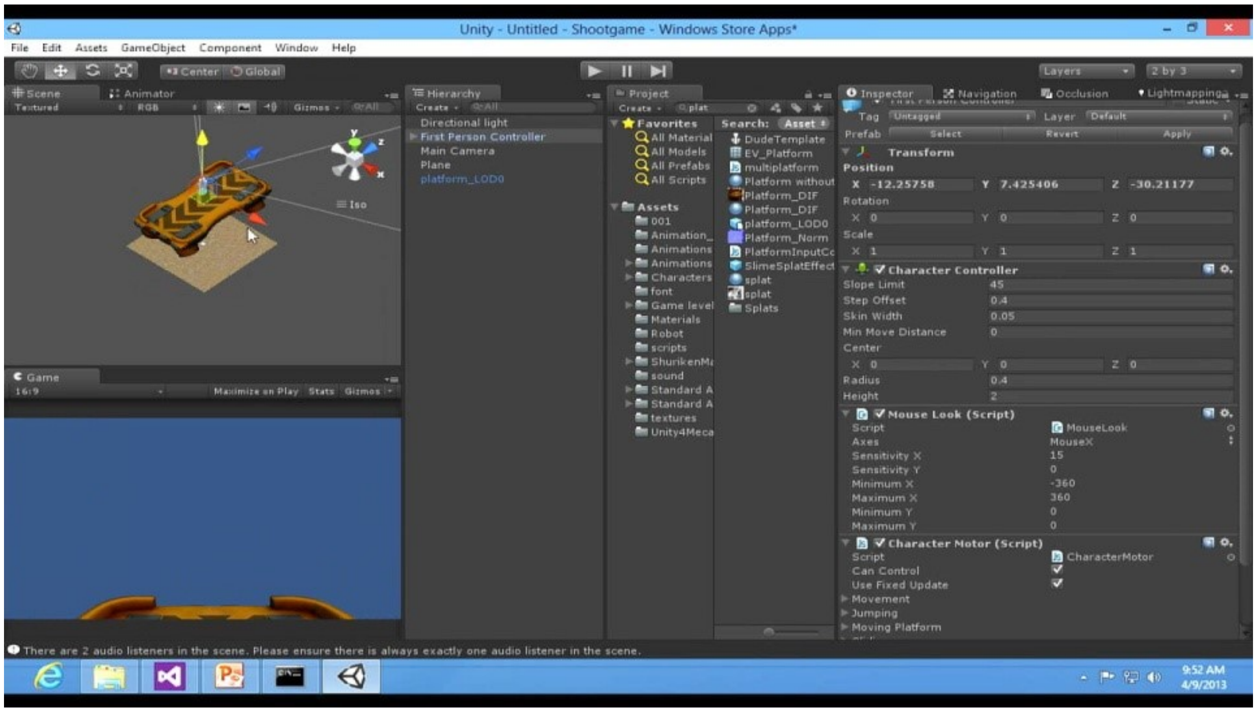
FIGURE 1.1. A screenshot of GameMaker running. (i.solidfiles.net)

Unity

Unity[1] is a 3D and 2D game development tool set. Unity is used by independent developers, as well as professionals. After art assets have been created, Unity can support the entire development process of a video game. Unity is itself the game engine that projects will run on. The engine can be run on a variety of platforms, which makes Unity an excellent tool for cross platform development. Games made with Unity consist of several

[1] More information on Unity can be found at http://unity3d.com/

objects, all controlled by user definable scripts. The scripts can be written in three standard languages: C#, JavaScript, or Boo. By giving users the power of a fully featured programming language like C#, developers can do almost anything with their Unity objects. Unity is not event driven out of the box, but users have created modifications that allow Unity to use an event driven game design approach. Unity's design is very modular and supports a powerful plugin system. Unity has a drag-n-drop interface that allows users to create and manipulate objects easily. A screenshot of Unity running can be seen in Figure 1.2 below.



Tag Untagged Layer Default
Prefab Select Revert Apply

Transform
Position
X -12.25758 Y 7.425406 Z -30.21177
Rotation
X 0 Y 0 Z 0
Scale
X 1 Y 1 Z 1

Character Controller
Slope Limit 45
Step Offset 0.4
Skin Width 0.05
Min Move Distance 0
Center
X 0 Y 0 Z 0
Radius 0.4
Height 2

Mouse Look (Script)
Script MouseLook
Axes MouseX
Sensitivity X 15
Sensitivity Y 0
Minimum X -360
Maximum X 360
Minimum Y 0
Maximum Y 0

Character Motor (Script)
Script CharacterMotor
Can Control ✓
Use Fixed Update ✓
Movement
Jumping
Moving Platform

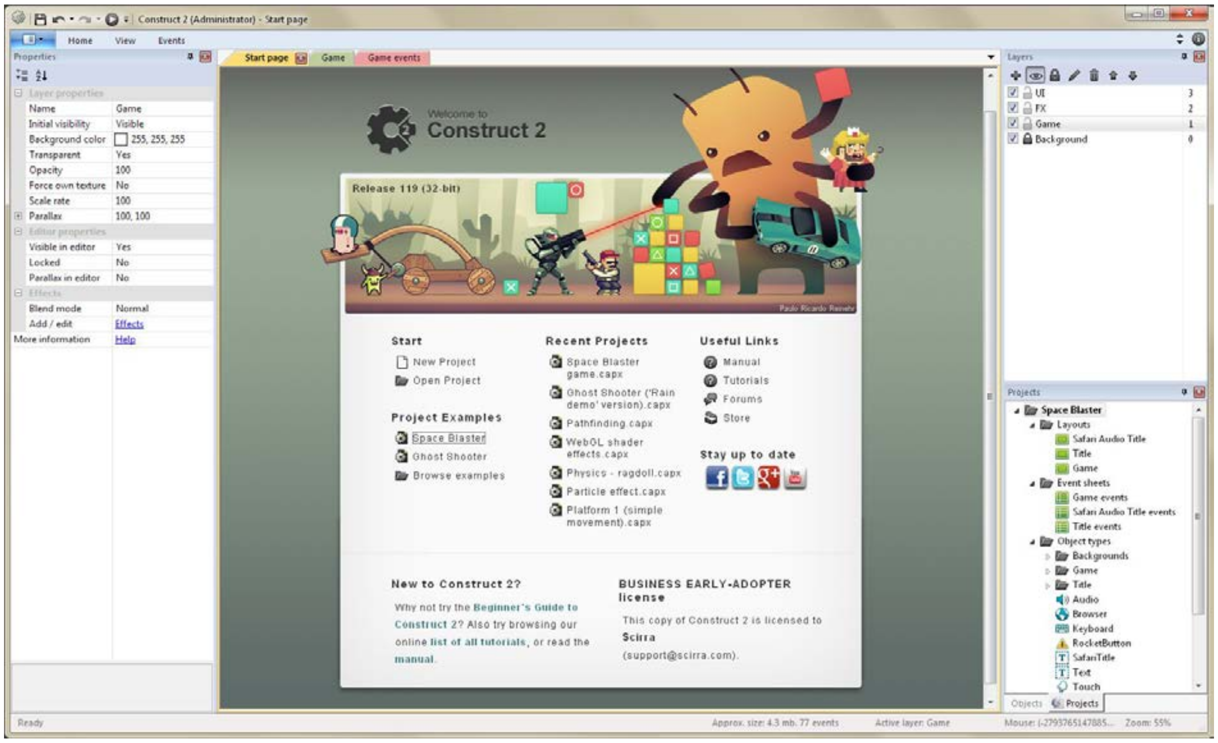
There are 2 audio listeners in the scene. Please ensure there is always exactly one audio listener in the scene.

9:52 AM
4/9/2013

FIGURE 1.2. A screenshot of Unity running. (video.ch9.ms)

Construct 2

Construct 2[1] is a 2D game editor targeted at hobbyist game creators. It is primarily a drag-n-drop editor, but also gives users a special programming language to control their game behavior with. The programming language is visually based, and therefore is not as general as standard languages such as C# or JavaScript. However, for advanced users, JavaScript can be injected into a Construct 2 project to control game behavior. Construct 2 games are event driven. Developers can specify event conditions and responses that control the action of the game. Construct 2 is built to create games that run inside HTML5's canvas technology. In this regard, Construct 2 games can be run anywhere that supports the canvas tag in HTML5. Construct 2 can be seen in Figure 1.3 below.



Layer properties	
Name	Game
Initial visibility	Visible
Background color	255, 255, 255
Transparent	Yes
Opacity	100
Force own texture	No
Scale rate	100
Parallax	100, 100
Editor properties	
Visible in editor	Yes
Locked	No
Parallax in editor	No
Effects	
Blend mode	Normal
Add / edit	Effects
More information	Help

New to Construct 2?

Why not try the Beginner's Guide to Construct 2? Also try browsing our online list of all tutorials, or read the manual.

BUSINESS EARLY-ADOPTER license

This copy of Construct 2 is licensed to Scirra (support@scirra.com).

FIGURE 1.3. A screenshot of Construct 2 running (static2.scierra.net).

[1] For information on Construct 2 can be found at https://www.scirra.com/construct2

1.3.2 Related Work Comparisons

After looking closely at GameMaker, Unity, and Construct 2, we were able to compare and contrast them. We gathered a set of features we wanted to replicate from each, and a set we wanted to avoid. The main metrics we used to measure each editor were:

- The way game mechanics are controlled.
- The level of visual editing.
- The range of platforms outputted games can target.

In terms of controlling game mechanics, Construct 2 is the odd one out, using visual programming as its primary way to manipulate game behavior. Unity and GameMaker both use a more traditional scripting approach. Between those two, Unity allows scripts to be written in standard languages, while GameMaker forces uses to use a proprietary language. We think that using a language that people are already potentially knowledgeable about is a good design choice. We decided to follow Unity's approach, and use C# for our scripts. There is already a plethora of documentation for C#, allowing us to focus our documentation efforts on the rest of our software.

All three editors had a very high level of visual editing. Drag-n-drop editing is present in all three editors, and is something that we wanted to include in Whiskey2D. The ability to configure levels visually is very appealing, and makes it much easier for the game creators to create their content.

Finally, all of the editors could create games for a variety of target platforms. Construct 2 achieved this by compiling games to HTML5 and JavaScript so that games could run in web browsers. The other two editors support game compilation to several different platforms. We wanted to support multiple target platforms as well.

1.3.3 MonoGame

In order to satisfy our goals, we decided to start with an existing library called MonoGame[1]. MonoGame is an open source implementation of Microsoft's XNA[2] library. XNA was a service that provides game programmers with a basic game loop, intermediate rendering capabilities, basic keyboard and mouse input, asset management, and other features fundamental to every game. XNA was also Microsoft's system to allow independent developers to push games to Xbox Live. XNA was deprecated on April 1st, 2014, roughly around the time Microsoft released the Xbox One.

While Microsoft hasn't publicly released anything to replace XNA, the open source community has delivered MonoGame, which provides most of XNA's features under the same interfaces. MonoGame is written with openGL and Mono instead of DirectX and .NET like XNA. By using openGL and Mono, MonoGame has the ability to deploy games to a variety of target platforms, such as Windows, Mac, or Linux. MonoGame has become an independent game developer standard, and has been used to create games like Fez[3], Bastion[4], and more.

We decided to use MonoGame because of the features it supports, as well as its ability to be cross platform. By taking advantage of MonoGame's feature set, we were able to focus on the engine architecture and editor rather than lower level requirements, such as

[1] More information for MonoGame can be found at http://www.monogame.net/
[2] More information on XNA can be found at https://msdn.microsoft.com/en-us/library/bb200104.aspx
[3] More information on Fez can be found at http://fezgame.com/
[4] More information on Bastion can be found at https://www.supergiantgames.com/games/bastion/

an openGL rendering and window management. Using MonoGame allowed us to focus on the implementing various features we found desirable in Unity, GameMaker, and Construct2. MonoGame was our starting point for Whiskey2D, and is the backbone of our system.

1.4 Roadmap

We built Whiskey2D's engine and the editor with our goals in mind and with inspiration from existing game editors. We implemented a fundamental set of game features, and editor features that will be discussed in Chapter 2. We'll explain how the engine and editor work, as well as the complete life cycle of a game made with Whiskey2D in Chapter 3. In Chapter 4, we'll go over a showcase game made with Whiskey2D called, Mechanical Plunge, and how it demonstrates the engine's features. In Chapter 5, we will explain how we worked as a team throughout the course of our MQP and the software development process we used. Finally, Chapter 6 discusses the challenges we faced in developing Whiskey2D, the lessons we learned, and how we were able to meet our project goals.

Chapter 2 - Functionality of Whiskey2D

Whiskey2D is an engine for 2D games. Every game built with Whiskey2D will consist of a few critical components, such as GameObjects and Scripts. In this Chapter, we will discuss the core pieces of a Whiskey2D game, and how designers can utilize Whiskey2D's engine. We will also detail the basics of the Whiskey2D user interface, as well as how it is used. Finally, we will explain how Whiskey2D supports a robust input feature that enables a full log/replay system. The next Chapter will discuss the technical concepts behind the features discussed in this Chapter. This Chapter is for the designer, while the next Chapter discusses Whiskey2D's underlying structure.

2.1 Core Concepts

Levels, GameObjects, Scripts, and Media are the core concepts to every Whiskey2D game. Whiskey2D has a simple game loop system that issues update messages to the current Level. A Level is the structure that includes a set of GameObjects. Every GameObject has some collection of Scripts. When a Level is active, all of its GameObjects get an update message. When a GameObject updates, it passes an update message to all of its Scripts. Fundamentally, GameObjects represent objects and their data, while Scripts represent object's behavior. Scripts can be added or removed at runtime, allowing for robust GameObject behavior patterns. Figure 2.1 below shows a breakdown of a Whiskey2D game into its basic parts.

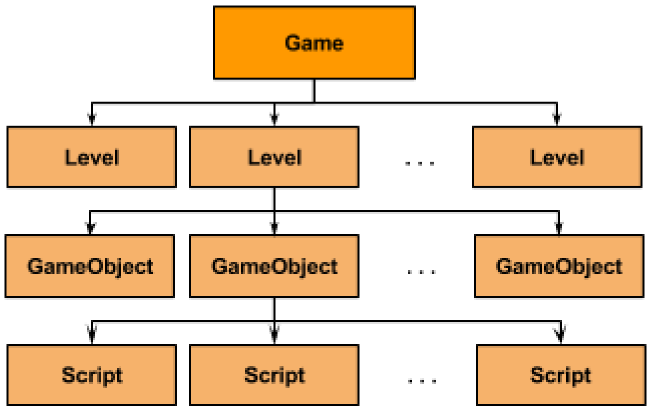


FIGURE 2.1. Basic breakdown of a typical Whiskey2D Game.

Designers make games by configuring GameObjects' properties, writing code for Scripts, and placing instances of GameObjects into Levels. Levels, GameObjects, and Scripts will be discussed in detail in the following sub-Sections.

2.1.1 Levels

All Whiskey2D games have a set of Levels. Each Level contains details about the world, such as GameObjects, a Camera, and visual information for the rendering process. One Level may be loaded at a time. When a Level is loaded, its *update()* function is called from the Whiskey2D engine. The *update()* function updates every GameObject, as well as the Level's Camera. The Camera can move around the world and zoom in or out, showing different parts of the Level to the User. A Level also has a background color and shader hints for Whiskey2D's rendering process. The game will always start at a given Level, and the designer can load different Levels at runtime.

2.1.2 GameObjects

A *GameObject* is a functionally-abstract class, where designers will implement subclasses for use in their game. For instance, a designer could create a *Person* class or a *Bike* class which are both subclasses of GameObject. These GameObject instances are the basic entities used to represent objects in a Whiskey2D game. The figure below (Figure 2.2) visualizes these possible GameObjects and how they would look in the Whiskey2D editor.

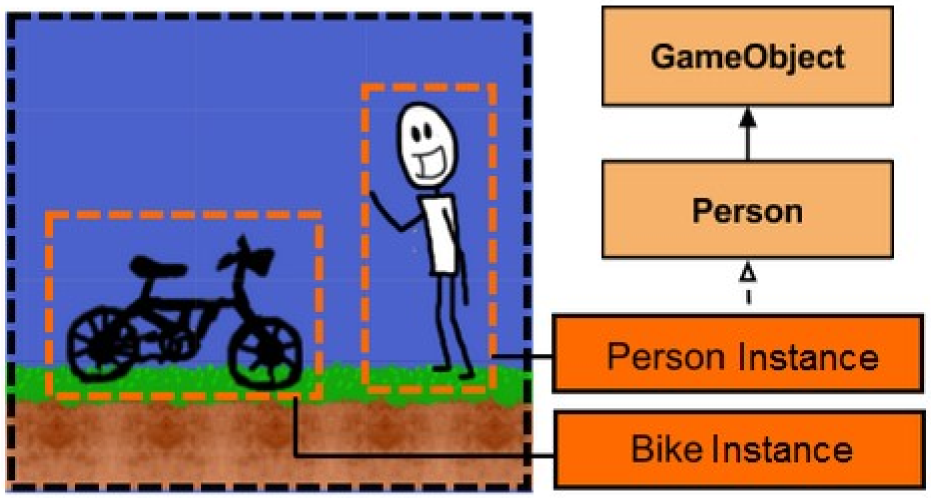


FIGURE 2.2. Instances of different GameObject subclasses (Person and Bike).

GameObjects contain properties that hold relevant information about the state of each object. By default, every GameObject comes with basic properties needed for the game engine, and include *Name*, *Position*, *Bounds*, *Sprite*, *Light*, *Layer, Shadows,* and *Active*.

As designers create their own types of GameObjects, they have the ability to add their own properties to each of their new GameObject types. For example, the designer that created the *Person* type might want to add a property *Age*, or *FavoriteFood*. Age could be treated as an integer while *FavoriteFood* could be a String, and now whenever a *Person* object is created, it will contain an Age and *FavoriteFood* property that can be accessed and changed as needed. Figure 2.2 below is a class diagram showing this connection.

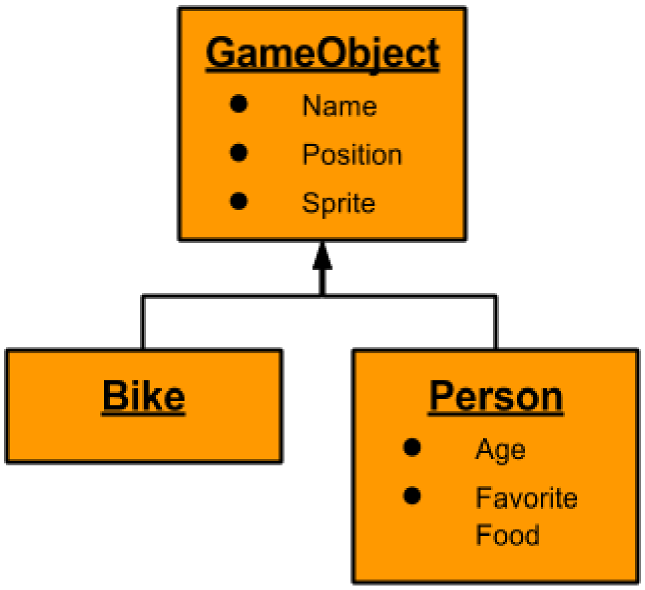


FIGURE 2.3. A class diagram for a Bike and Person.

As stated earlier, these GameObjects are what make up a Level, and their properties are updated on every one of the game's update ticks. The individual GameObjects contain a set of *Scripts*, which are the entities that actually update the object's property values. It is the interaction between GameObjects and their Scripts that make the game play and change through the passing of time.

Shared Objects

Whiskey2D comes with a few GameObject types that are built into the engine, and are available for use in every project. This includes *CameraMaster*, an object used to control the camera, *CamZone*, a camera zone object, *SimpleLight*, a light object, and others. They are built into Whiskey2D's engine and therefore cannot be edited. A complete listing of shared objects can be found in the designers' guide in Appendix A.

2.1.3 Scripts

A Script is an abstract class that allows designers to control GameObjects. Designers can implement subclasses of Script and attach them to their GameObjects. Every Script must have an *onStart(), onUpdate(),* and *onClose()* method. The *onStart()* method is called when the Script is first attached to a GameObject. *onUpdate()* is called every time the Script's GameObject is updated by the Level. Finally, *onClose()* is executed when the GameObject is closed, or terminated from the Level. Every Script may also be active or inactive. If the Script is inactive, then it will not receive any calls to *onStart(), onUpdate()*, or *onEnd()*. Since designers can write whatever code they want in the abstract functions, they can activate or deactivate Scripts at any time.

The abstract Script class has a reference to the GameObject that it belongs to. The GameObject is exposed to the designer's subclass, allowing them to access and control properties of the GameObject. The Script requires that the subclass be coded for a specific type of GameObject. By ensuring that each type of Script only operates on a single type of GameObject, the correct properties and type information of the GameObject can be given to the Script at compile time. Below is a class diagram (Figure 2.4) for Script and our example implementations.

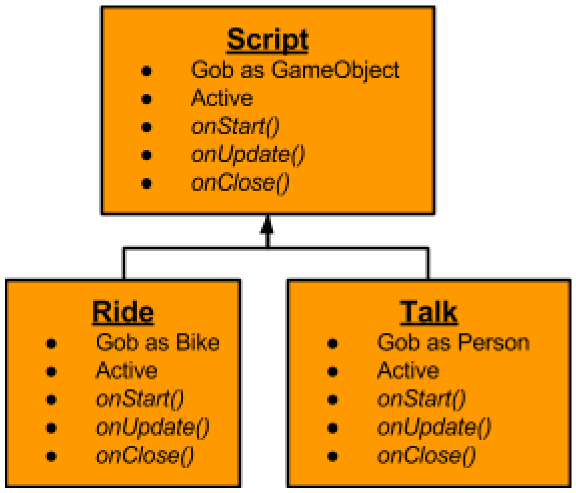


FIGURE 2.4. A class diagram for Script, Ride and Talk.

Following the example described in the previous Section, a designer might write a *Ride* script for the GameObject, *Bike*, and a *Talk* script for the GameObject, *Person*. The *Talk* Script would have the required three methods, *onStart(), onUpdate(),* and *onClose(),* as well as a class property, *Gob*, that was of GameObject type, *Person*. The *onStart()* function could contain code that initializes the Script, and the *onUpdate()* method could have code to make the person say their *Age*, or *FavoriteFood*. One possible implementation of this behavior can be seen in the code sample (Figure 2.5) below.

```
public class Talk : Script<Person> {

        bool hasSpoken; //local variable

        public void onStart(){
                hasSpoken = false; //initialize variables
        }

        public void onUpdate(){
                if ( !hasSpoken ) {
                        //only speak if script hasn't spoken yet
                        //GameManager.Log will be discussed in Chapter 3.
                        GameManager.Log.debug("I am " + Gob.Age + " years old.");
                        hasSpoken = true;
                }
        }

        public void onClose(){
                //do nothing. Required by Script class.
        }
}
```

FIGURE 2.5. A code snippet showing the basic structure of a script.

Shared Scripts

Like *Shared Objects*, Whiskey2D has built-in Scripts that are available to every Whiskey2D project, and made to be used with some of the Shared Object types. These scripts give behavior to the Shared Objects, and help facilitate rapid development of games for the designer. However, not every Shared Object has a built-in Shared Script. These Shared Scripts include the *CameraMasterControl*, used to control the camera, *SpriteEffectControl*, used to control different SpriteEffect objects, and *TriggerActivate*, which controls the behavior of a TriggerZone object.

2.1.4 Designer API

Whiskey2D's engine provides a range of services to designers. When a designer is writing a Script, they can access Whiskey2D's services by making calls to the GameManager. The GameManager is a centralized location where the designer can obtain information about GameObjects, User Input, Levels, Rendering, Resources, and Logging. A GameManager Application Program Interface (API) guide for writing Scripts can be found through Appendix A.

The GameManager is also the component that issues update calls to Levels, and subsequently, all GameObjects and Scripts. The GameManager is ultimately responsible for everything that happens in a Whiskey2D game. Chapter 3 will go into further detail on the GameManager and its properties.

2.1.5 Media Support

Whiskey2D supports two main types of media, sounds and images. All media must be stored in the game's *media* directory. Any media that is in this directory becomes accessible during game time, and the designer can use it to bring their game to life.

Sound

Sound is a critical component of every game. Currently, Whiskey2D supports many flavors of .wav audio. Once a sound file is placed in the media directory, it can be loaded by creating a new Sound object with the .wav's file path. Sound objects must be controlled through a Script, or the shared object, SoundZone. Sounds can be played, paused, stopped, or looped. Scripts may adjust a Sound's pitch, volume, and pan, allowing designers to create complex soundscapes in game. To play a sound over itself, like a machine gun's echo playing over previous gun shots, a Sound object can duplicate itself to play the sound again. The details of the Sound class can be found in Appendix A.

Images

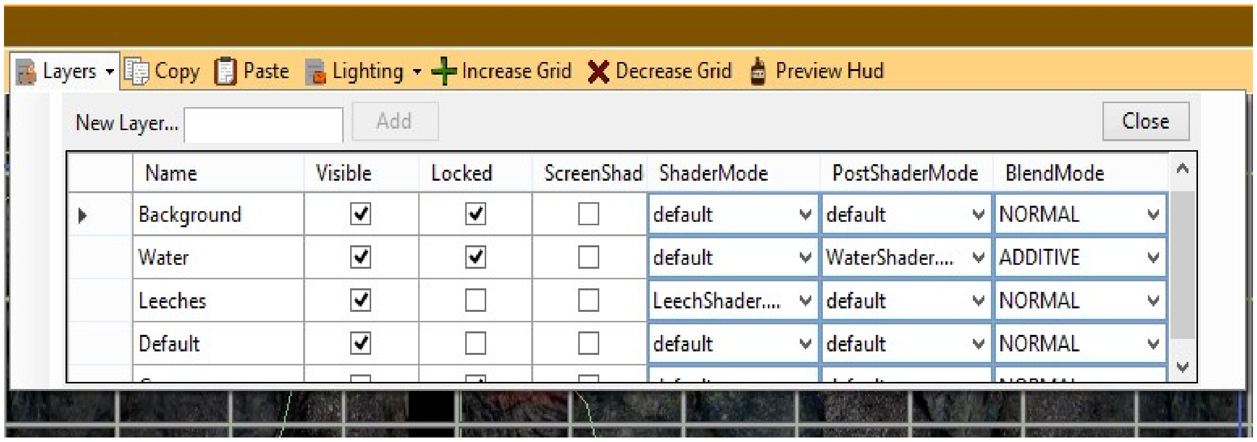
Whiskey2D allows basic geometry to be created dynamically, so designers can create entire games without ever having to import an Image. However, having Images helps bring games to higher level of realism, and are incredibly important. Images are loaded as Sprites into Whiskey2D. A Sprite contains information about the image, as well as how the designer has specified the image to display in game. The Sprite can be scaled, colored, and rotated. When the Sprite is scaled, it can stretch the image, or it can be set to tile the image. In addition, if a designer uses a sprite sheet as the image for a Sprite, then the Sprite can be told to display only certain areas of the sprite sheet in rapid succession, thus achieving 2D animation. By default, every Sprite is set to display the entire image, by assuming that the image is split into a 1x1 grid. However, when a sprite sheet is used, the designer can change the Rows and Columns properties of the Sprite to specify the image grid. The Sprite can still be scaled, and rotated correctly even when the grid is not 1x1. More details on the precise commands for Sprite can be found in Appendix A.

2.1.6 Custom Rendering

Whiskey2D supports a wide variety of ways to customize the rendering process. There are options to change the global level effects, the lighting and shadowing effects, and even ways to specify custom shaders for individual rendering layers.

All GameObjects have a Layer that they are drawn on. The Layer has a blending option, and two shader options, ShaderMode, and PostShaderMode. The Layer view can be seen in Figure 2.6. The blending option will determine how GameObject's Sprites blend over each other. By giving the Layer a ShaderMode, it will apply the supplied shader to every GameObject as it draws. The PostShaderMode will apply a shader to the entire

Layer, after all GameObjects have been drawn. All of these settings can be adjusted from the Layer button, located on the Level's toolbar. New shaders can be created inside the media directory.



Name	Visible	Locked	ScreenShad	ShaderMode	PostShaderMode	BlendMode
Background	✓	✓	☐	default	default	NORMAL
Water	✓	✓	☐	default	WaterShader....	ADDITIVE
Leeches	✓	☐	☐	LeechShader....	default	NORMAL
Default	✓	☐	☐	default	default	NORMAL

FIGURE 2.6. The Layer view.

GameObjects also have lighting and shadowing settings that can be adjusted in their properties. Every GameObject can emit light, and a can cast shadows. There are several options for customizing the behavior of shadows that can be seen in the Figure 2.7 below.

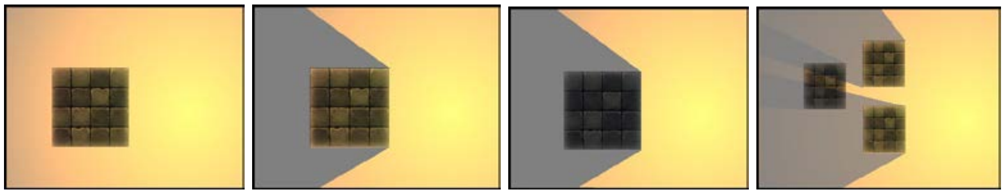


FIGURE 2.7. Various shadowing configurations. In order from left to right, *no shadow, shadow and light inclusion, shadow and light exclusion, partial shadow and light inclusion.*

Additionally, the entire Level has customizable global effect settings. The designer can control the level of bloom, saturation, and more from the level properties. Various setting configurations can be seen in Figure 2.8 below.

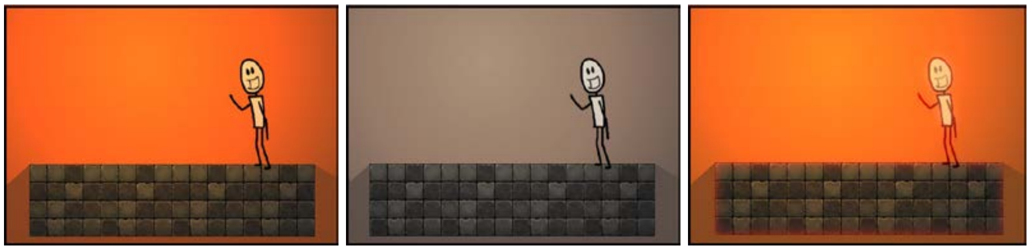
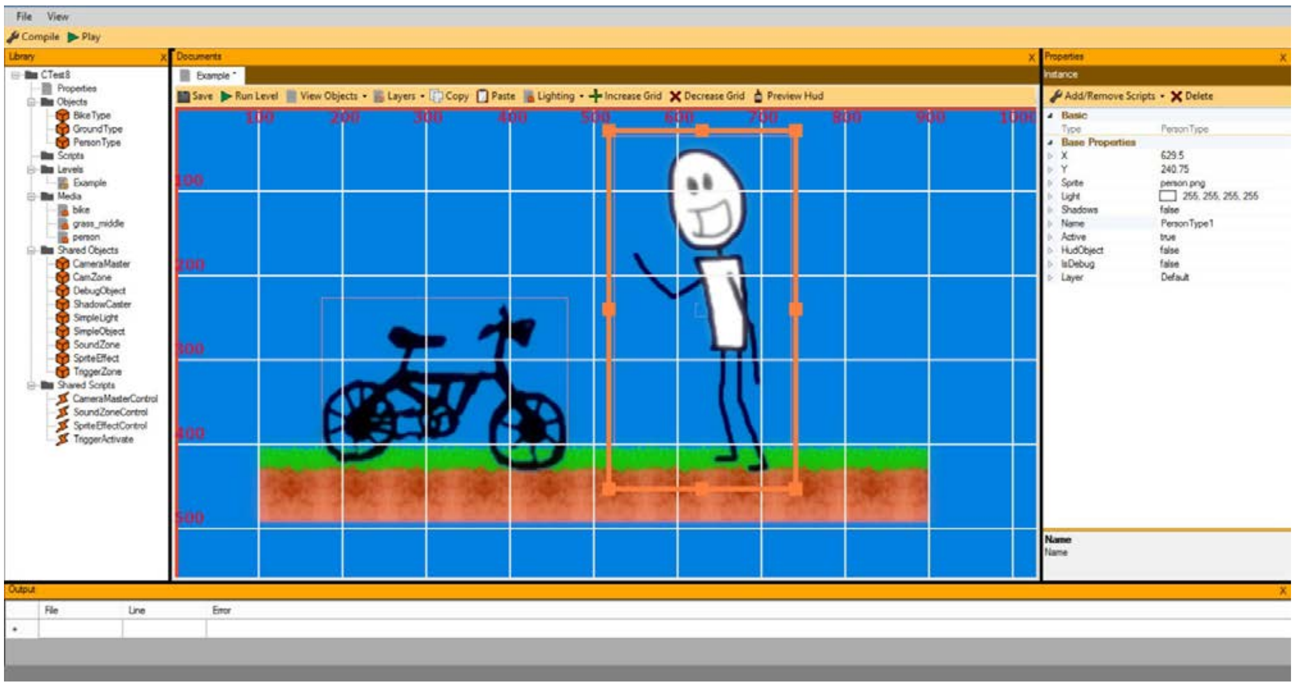


FIGURE 2.8. Various bloom configurations. In order from left to right, *no bloom, desaturated no bloom, saturated bloom*

2.2 Editor Concepts

The Whiskey2D editor takes its inspiration from other popular tools and IDEs like Eclipse or Visual Studio. Like Visual Studio, only one project (a solution in Visual Studio) may be loaded at a time. There are several views in the editor, each exposing different aspects of the Whiskey2D game in development. A sample of the Whiskey2D editor can be seen below, in Figure 2.9.



Basic	
Type	PersonType
Base Properties	
X	629.5
Y	240.75
Sprite	person.png
Light	255, 255, 255, 255
Shadows	false
Name	PersonType1
Active	true
HudObject	false
IsDebug	false
Layer	Default

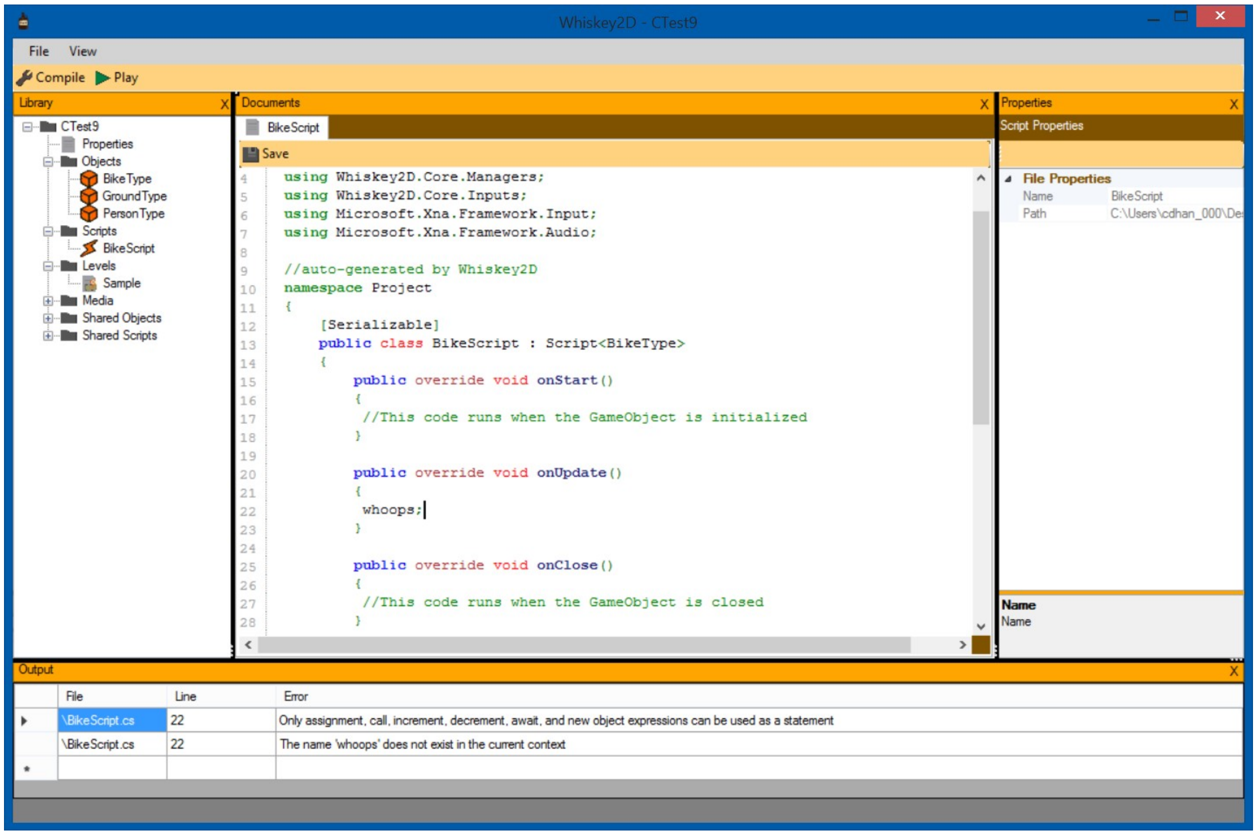
FIGURE 2.9. An instance of a Player, a Bike, and a Ground object

The Library View, on the left, shows all of the files associated with the game. It shows the project directory's main folders as Types, Scripts, Levels, Media, and common elements of every Whiskey2D game. By selecting a file, designers can open a new tab in the center view, or the Documents View in the center, as well as setting the Property View on the right.

The Document and Property views are responsible for showing the details of the selected file. In the screenshot above, a Level file has been selected. The Document View is showing the contents of the Level. The designer has placed a few GameObjects, and selected the Person instance. The Property View is showing the details of the selected instance, the Person.

In the screenshot below, Figure 2.10, the designer has created and selected a Script from the Library called BikeScript. The Document View has added a second tab, and now shows the Script's code, and the Property View shows the details of the Script. Notice that the Script's code contains errors. The errors are reported from the Output View on the bottom of the screen.

Every tab in the Document View, as well as the Property View has a toolbar of commands at the top. The commands in the toolbar are sensitive to what kind of information is being shown. For example, when a Script tab is being shown, the only command available is *save*, but when a Level tab is shown, commands related to the Level become available. The top of the screen also has a command bar, with *Compile* and *Play* as the only commands. These commands are accessible no matter what is being shown elsewhere. They can be used to test the game. Lastly, the menu strip at the top can be used to create new projects, open previous projects, and create new files.



```
using Whiskey2D.Core.Managers;
using Whiskey2D.Core.Inputs;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Audio;

//auto-generated by Whiskey2D
namespace Project
{
    [Serializable]
    public class BikeScript : Script<BikeType>
    {
        public override void onStart()
        {
         //This code runs when the GameObject is initialized
        }

        public override void onUpdate()
        {
         whoops;
        }

        public override void onClose()
        {
         //This code runs when the GameObject is closed
        }
```

Name	BikeScript
Path	C:\Users\cdhan_000\De

File	Line	Error
\BikeScript.cs	22	Only assignment, call, increment, decrement, await, and new object expressions can be used as a statement
\BikeScript.cs	22	The name 'whoops' does not exist in the current context

FIGURE 2.10. Errors appear in the bottom of the UI.

As mentioned, the Whiskey2D editor is similar to other editors, such as Eclipse or Visual Studio. Whiskey2D designers should be familiar with the concept of a project library and having several tabs open at once. Whiskey2D's UI is simple and straightforward, yet powerful and robust.

2.2.1 Using the Editor

There are a many steps that must be taken to create a game in Whiskey2D. When starting a new project, the designer starts with a blank slate. The only things that are given are shared objects that can be used later in the designer's levels.

The first thing the designer must do in starting a new project is to create their first Level. A *Level* will have a designer-entered name, as well as a background color and a camera position. The level will act as a container for a group of objects that a game will contain. Conceptually, Levels will be loaded at a certain point in response to some state change in a game. A Player might do something while playing a game which prompts a change in what objects are present, the current location of the game, its music, the background, and so on.

After the designer creates her first Level, she can now populate her Level with GameObjects. These objects make up everything that a level has in it, such as the main character, any collidable objects, and any entities in the game that can interact with the other objects. By default, the designer can instantiate any of the shared objects, but a designer will most likely will want to create her own type of objects, too. To create a GameObject type, the designer must provide a type name. The type will come with all the default properties that all GameObjects contain, but the designer can then add any other relevant properties that can be useful in the game. The designer must also decide what the object will look like in-game. This is found as a property of the object, called Sprite.

Once the new GameObject types have been created, the designer can drag in instances of these objects into the level. These objects can be placed into the level at the position where the designer initially wants them to exist. These objects can be set to active or not active depending on whether the designer wants them to exist as soon as the level is loaded or whether they want the object to spawn in response to another action.

Now that the level has GameObjects populating the game, the GameObjects need scripts to define their behavior. A designer will create and implement scripts for a type-by-type basis, meaning each script is written for a specific type of GameObject, and only will work for an object of that type. A script can be used, for example, to move a playable character in the game in response to a user's input. Once the Scripts are implemented, the designer can add these scripts to individual *GameObject* instances of the correct type.

With the level now containing game objects and the game objects now containing their scripts, the level can be played by a user or tested by the designer. Once the scripts are refined to work as they should, and the game objects are fleshed out, the designer can go forth and make another level. These levels can be loaded by scripts that game objects contain. Before another level is loaded, the current level must be unloaded. Once the project contains more than one level, the designer must select what the starting level must be for the game. Once the starting level is selected by the designer, whenever the game is launched, the selected level will be run first. Sequential levels will be launched through scripts, in response to some state being achieved.

2.3 Input, Logging, and Replay

A very important element in Whiskey2D is its built-in log and replay service. Every Whiskey2D game generates a log file that can be used to replay a gameplay session. Replays can be used to send bug reports to the designer, to share memorable moments of gameplay, or could be evolved slightly to support live debugging of a game. The details of how Replay is supported architecturally will be discussed in the next chapter. However, the rest of this Section serves as a brief feature-overview of log and replay.

Players of Whiskey2D games may enter input through a keyboard or mouse. Whiskey2D retrieves the Input from the hardware, and makes it available to designers' Scripts. From Scripts, designers can handle various input actions. Designers can also send messages to Whiskey2D's logging service, which are stored in a log file next the game's executable. The designer's messages can also be seen in-game as well. Logging is an invaluable service for designers that allow them to debug code.

The log file contains all of the designer's log messages, as well as a complete set of input messages. Whiskey2D internally sends log messages detailing the Input being received from the keyboard and mouse. By storing all of the input in the log file, the log file can be used as a replacement keyboard and mouse. To be clear, the log file that contains designer messages is the same file that contains all input messages. As Figure 2.11 shows, Whiskey2D's input source may be either attached hardware, or it may be a log file containing all of the relevant input information.

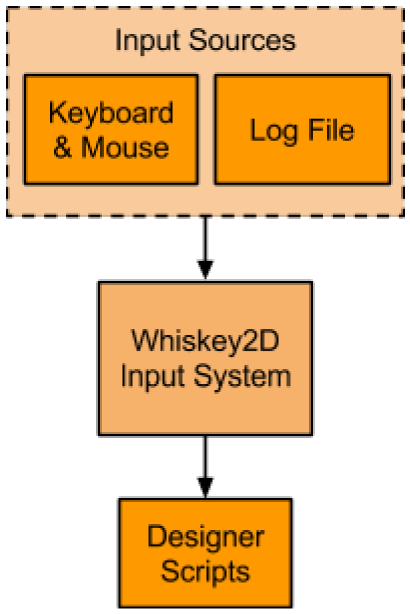


FIGURE 2.11. A process diagram showing that input can come from either physical hardware or a log file

By loading a log file as input to Whiskey2D, Whiskey2D will effectively replay a game exactly how it was originally played. Since the engine is deterministic, the input is the only thing driving the game in a unique direction. Essentially, the Input and Logging services are built in such a way that entire Sections of gameplay can be exactly replicated at a later date, just by loading the original log file. There are certainly limitations to this feature, such as networking or file access. However, despite the limitations, the Replay feature gives Whiskey2D games a powerful feature.

Chapter 3 - Architecture of Whiskey2D

Whiskey2D's architecture comprises of three primary components, the Core, the Editor, and the Launcher. As discussed in Chapter 2, the designer constructs a game in the Editor, and then compiles it into a game that runs inside the Core. The third component for running the game, called the Launcher, is used to launch a designer's compiled game. The three primary components can be seen in Figure 3.1. In this Chapter, we will discuss the Core, the Editor, and the Launcher.

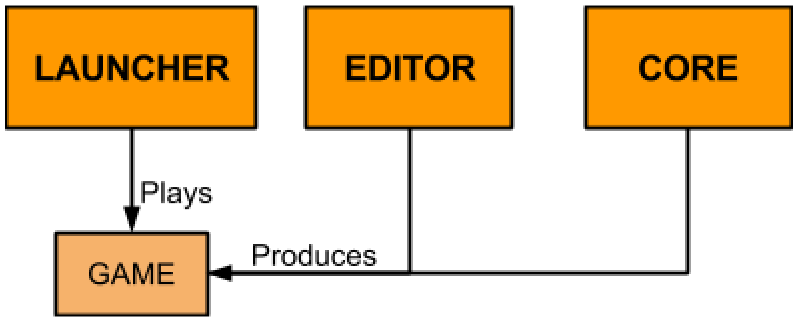


FIGURE 3.1. Whiskey2D's three major components are shown.

3.1 Core

The Core is the code that runs a designer's game. It is where all of the fundamental classes exist that support the Editor and a designer's game, such as Level, GameObject, and Script. Everything that happens in a designer's game is defined by their project code, and the Core. The backbone of the Core is the GameManager. It managers all of the services vital to running a designer's game.

3.1.1 Game Manager

The GameManager's primary responsibility is to run the Game Loop. The Game Loop updates all of the Core's critical services, and renders all GameObjects. The Game Loop, seen in Figure 3.2, is akin to an organism's heart, and is what sets the game's rhythm. The game's data is only updated on each of the Game Loop's iterations. In the first step of the Game Loop, Input is retrieved from either the keyboard, or a log file. Secondly, the Logging system records the input. After the input has been logged, the Game Loop sends an update message to the Object Manager. This update message will update all of the active Game Objects and Scripts. Finally, the Game Loop gives control to the Render Manager, and the game is drawn to screen. Each of these processes will be discussed in greater detail below.

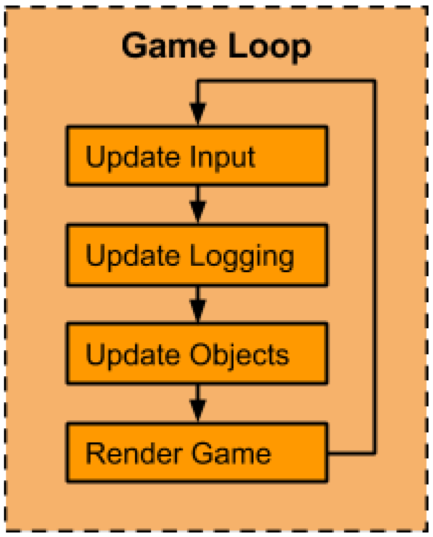


FIGURE 3.2. A process diagram showing the game loop stages

In addition to running the Game Loop, the GameManager provides designers access to all of the Core services, such as Input, Object management, Level data, and Logging, as seen in Figure 3.3. The GameManager is a singleton, and has a set of static accessors that retrieve the services.

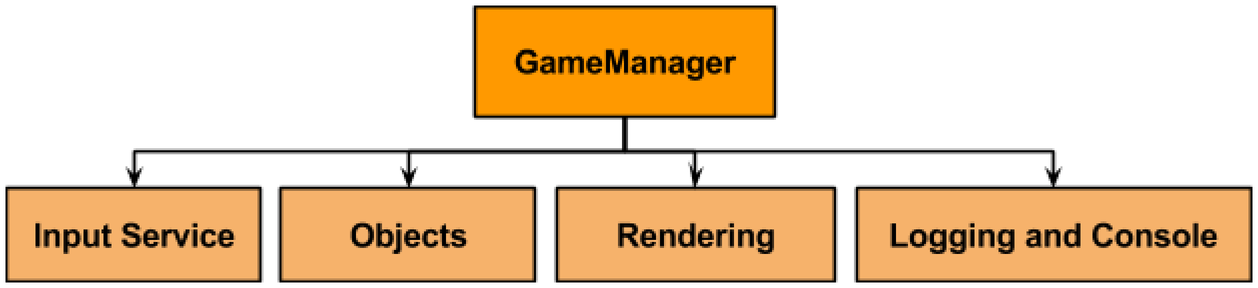


Figure 3.3.: A breakdown diagram of the GameManager's services

3.1.2 Game Services

Input and InputSource Service

The input service is provided by the InputManager and the InputSourceManager in the Core. The InputManager processes the incoming input. The InputSourceManager manages where the input is coming from. Whiskey2D is built so that different types of input sources can be swapped out during gameplay, but functionally work the same. Because of this, Whiskey2D can use either input from a mouse and keyboard or a log file. The data sent from one input source looks the same as any other input source to Whiskey2D.

Logging and Replay Service

The LogManager provides Whiskey2D with a logging service that runs whenever the game is run. The LogManager writes all the input coming from the current input source to a log file. This log file is created in such a way that it can be used as an input source itself. This means that after a player has played a portion of a level, the user can "restart" their level and switch their input to the log file, which in turn makes the game play from the file. The log file contains all of the input choices that the user made in the level previously, and thus the game will play identical to how the human player did before. In order to support replay of randomly generated values, Whiskey2D uses a custom Random service. The Random service logs its original seed in the log file so that the same set of random numbers can be produced on each replay.

The LogManager can also write other types of messages to the log file. These messages, called Debug messages can be used by a designer in their Script files to debug any problems they are having, or to check a variable's values at specific times or after a certain action is made.

Console Service

The designer can access a variety of in-game debug features of Whiskey2D by pressing '~' and opening the development console. Opening the console will pause the game, and allow the designer to run several commands. The current command set is:

●	*help*	*displays all available commands to the Console*
●	*exit*	*closes the game*
●	*replay*	*initiates a replay of the game*
●	*reset*	*restarts the game*
●	*get*	*reads a value of a GameObject's property.*
●	*countobjects*	*displays the number of existing GameObjects*

Adding commands can be done easily, by creating a new subclass of ConsoleCommand. Unfortunately, commands may only be added at the Core's compile-time, and are therefore not modifiable later.

Object and Levels Services

The Object service is provided through ObjectManagers. ObjectManagers contains and manage lists of GameObjects. In order to instantiate a GameObject, an ObjectManager must be given the GameObject's constructor. The GameManager has an ObjectManager called Level, which is the default ObjectManager for GameObjects.

The ObjectManager groups the GameObjects into four lists: *newlyAdded, active, inactive*, and *dead*. When a GameObject is added to the ObjectManager, it is put into the *newlyAdded* list. Later, the GameObject will be moved to the *active* or *inactive* lists. Finally, a GameObject will be placed in the *dead* group when the GameObject is closed.

When the ObjectManager updates, it follows three basic steps. First, it moves all GameObjects in *newlyAdded* to the *active* or *inactive* lists depending on if the GameObject should be available at the start of the Level. Next, the ObjectManager issues update messages to every GameObject in the *active* list. Each GameObject in this list may create or close other GameObjects. Doing so does not immediately affect the ObjectManager's *active* list. Finally, the ObjectManager destroys every GameObject in the *dead* list and clears it. Using the different lists of GameObjects allows the GameObjects to create and remove GameObjects. The ObjectManager also provides several utility functions to access GameObjects. Notably, the ObjectManager can fetch type specific GameObjects by name.

A Level is a subclass of ObjectManager. A Level contains all of the initial GameObjects required to play. In addition to providing the ObjectManager features, a Level also contains various visual information, such as global lighting properties, shader properties, and Camera data.

Rendering

Whiskey2D's rendering process is invoked every tick of the Game Loop, immediately after all of the GameObjects have been updated. The rendering is controlled by the RenderManager, and relies heavily on MonoGame. Figure 3.4 shows the main phases in rendering.

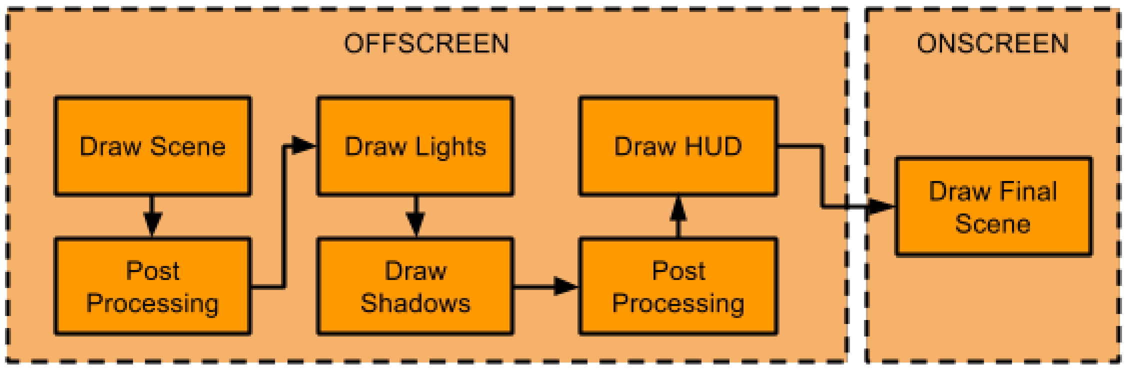


FIGURE 3.4. A process diagram, showing the various rendering stages.

In the first phase of rendering, all GameObjects have the opportunity to draw their image to an intermediate buffer, called the scene buffer. This process is actually quite complex, as the GameObejcts are first rendered to a buffer for their layer. As the GameObjects are rendered to the layer buffer, any shader effect on the Layer will be applied to the GameObject. Once all the layer's GameObjects have rendered to the buffer, the layer's post shader effect will be applied to the entire buffer. This process is repeated for all layers. As each layer finishes rendering, their buffer is written to the scene buffer. Finally, after all layers are rendered to the scene buffer, on a separate buffer, all GameObjects can draw to a lightmap. The lightmap is taken, shadowed, and then multiplied onto the scene buffer, creating a lit world. The lighting and shadowing system is customizable.

Before the shadowed lightmap and the scene buffer are combined, each are run through a post processing shader pipeline that enables bloom, blur, and saturation. Again, these settings are highly configurable.

The Heads-Up-Display (HUD) objects and the in-game console are rendered to another offscreen buffer. While the lightmap and the scene buffer are both rendered using the active Level Camera's transform matrix, the HUD buffer is rendered directly, with no transformations applied. Finally, the lit scene buffer and the HUD buffer are layered over one another, and drawn to the screen. The two main steps of the lighting rendering process can be seen in Figure 3.5 below.

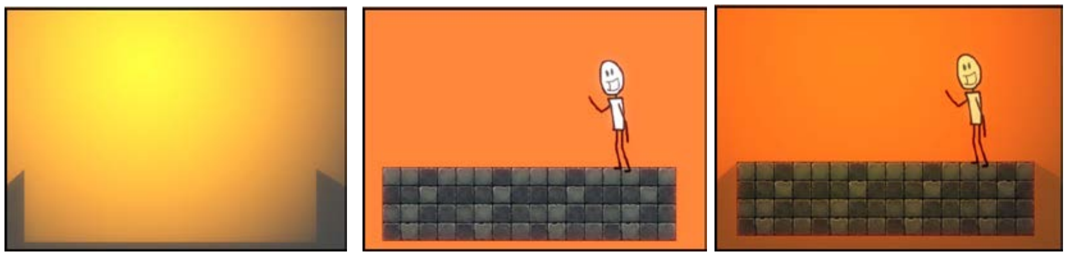


FIGURE 3.5. The phases of rendering lights and shadows. Left: only lightmap is rendered, with the shadowmap mask applied. Middle: the regular scene. Right: images are combined.

3.2 Editor

The Whiskey2D Editor is the system that allows designers to visually create games. The Editor can be thought of as three primary subsystems working in tandem. First, there is a file system that the Editor must maintain. Secondly, there is a large system in place to model a complete Whiskey2D game, without ever compiling any code. This system uses Descriptors, which will be discussed in Section 3.2.2. Lastly, there is the entire User Interface (UI) that supports the previous two systems. Figure 3.6 shows these main components of the Editor.

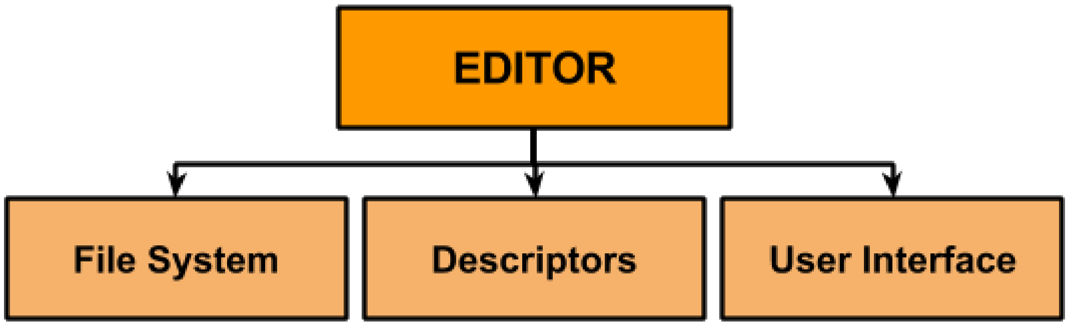


FIGURE 3.6. A breakdown of the Editor's primary components.

3.2.1 File System

At any given moment, the Editor has two file directories to manage. Primarily, the Editor must maintain its own working directory, where the editor executable is stored. The other directory is the Editor's currently loaded Project directory. Every Project that the designer works on is stored in a unique directory. Both the Editor's directory and the Project directory are vital to the game creation process, and can be seen in Figure 3.7.

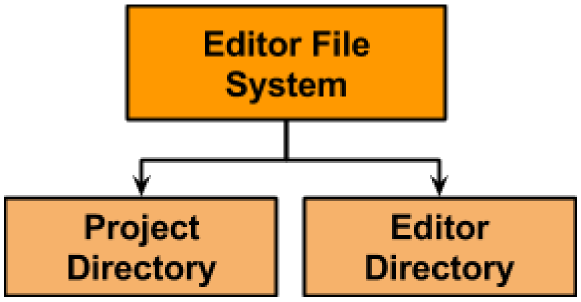


FIGURE 3.7. A breakdown of the two directories maintained by the Whiskey2D Editor

Project directory

The Project directory contains all of the information about a designer's Project. When a designer creates a new Project, Whiskey2D will generate the directory structure seen in Figure 3.8.

In the top level of the Project directory, there are two files, *.gamedata* and *.whiskeyproj*. The *.gamedata* file will be discussed in greater detail in Section 3.3.2, but generally can be thought of as a file containing all of the game's uncompiled data. The *.whiskeyproj* file is the main file of the Project. When a designer wants to load an existing Project, they are prompted to select a *.whiskeyproj* file. The file contains data about the project, such as title, and graphical settings.

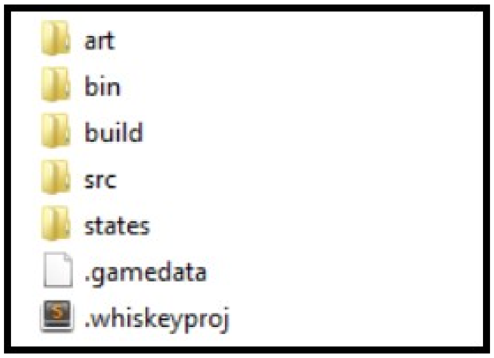


FIGURE 3.8. The project directory.

The *art* directory contains all of the media that a designer wants to include in their game. The details of how the media is handled will be discussed in the next Section on the Editor's working directory.

The *bin* directory contains the code library that is generated when the designer builds their game. The only file in the *bin* directory will be gameData.dll, and will be discussed in greater detail in Section 3.3.2.

The *build* directory is where the generated game is placed. The *build* directory will contain all of the files required to play a Whiskey2D game, including all of its dependencies.

The *src* directory contains all of the designer's generated C# source code. When a designer creates a new type of GameObject, or Script, a file modelling the behavior of the newly created entity is made and saved in the *src* directory. The generated code may be edited with any text editor program. Each generated file has a code segment blocked off, with comments asking designers to not edit the marked section. Other than the marked off section, a designer can make any changes to a generated file they'd like.

The *states* directory contains all of the designer's Levels. Each Level is represented by a *.state* file, and contains the data for every detail about a level.

Editor Directory

The Editor directory is where the Editor executable runs from. This directory is generated by Visual Studio when Whiskey2D is built. The directory structure can be seen in Figure 3.9 but is not as important as the previously discussed Project directory. The critical pieces of the Editor's directory are the sub-directories*, compile-exe, compile-lib,* and *compile-media*.

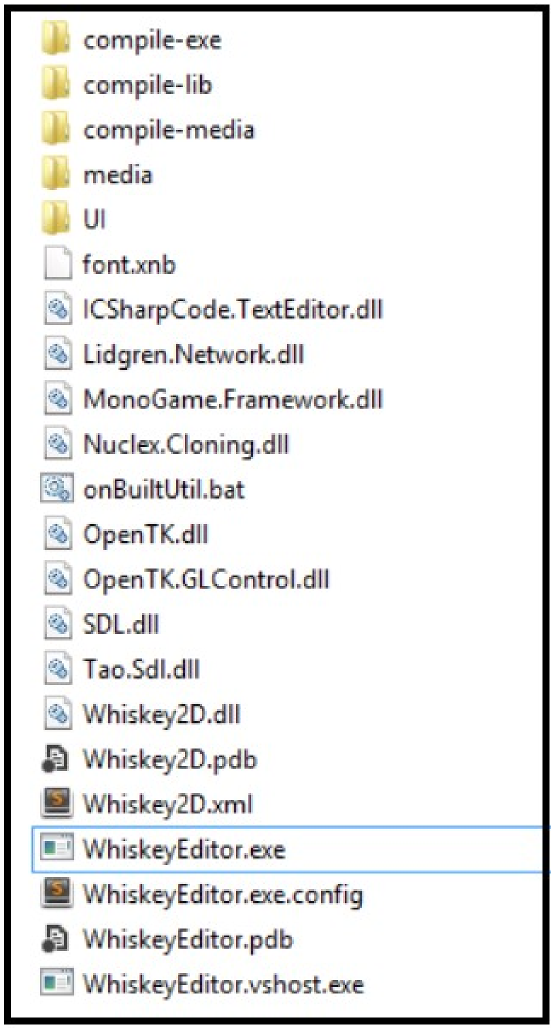

compile-exe
compile-lib
compile-media
media
UI
font.xnb
ICSharpCode.TextEditor.dll
Lidgren.Network.dll
MonoGame.Framework.dll
Nuclex.Cloning.dll
onBuiltUtil.bat
OpenTK.dll
OpenTK.GLControl.dll
SDL.dll
Tao.Sdl.dll
Whiskey2D.dll
Whiskey2D.pdb
Whiskey2D.xml
WhiskeyEditor.exe
WhiskeyEditor.exe.config
WhiskeyEditor.pdb
WhiskeyEditor.vshost.exe

FIGURE 3.9. The directory for the editor.

The *compile-lib* directory contains all of the necessary library files for a Whiskey2D game to run. The majority of these libraries are the MonoGame libraries. The *compile-exe* directory contains a copy of the Launcher.exe. The Launcher.exe will be discussed in greater detail in Section 3.3.3. The contents of both *compile-lib* and *compile-exe* are copied to the Project directory's *build* directory when the designer builds a game.

The other important directory is *compile-media*. When a designer loads a media asset into their project, the asset is copied to the Project's *art* directory, as well as the Editor's *compile-media* directory. The reason for this is because the Editor's working directory is not the same as the Project directory. When the Editor needs to load an asset, it must be given a path relative to its own working directory. This is a limitation from the underlying MonoGame code. To overcome the limitation, the asset is simply copied to *compile-media,* so that the Editor can load it relative from the working directory. When a designer builds their game, only the media from the Project's *art* is copied to the outputted *build* directory. Nothing from *compile-media* is used in the build process.

3.2.2 Descriptors

Descriptors are the second primary subsystem of the Editor. A Descriptor is anything that models something that has yet to be created. The important concept behind Descriptors is that they can generate the data they are modeling. All of the designer's GameObjects and Scripts need to be modelled by Whiskey2D's Editor. This modelling is achieved through using specialized Descriptors for each main kind of data in a Whiskey2D game.

Everything that the designer builds in the Editor is stored as a Descriptor. A Descriptor can be instructed to generate the structure it models. Using Descriptors, Whiskey2D can model type information about GameObjects and Scripts without ever having to compile anything. Descriptors are beneficial because they allow Whiskey2D to avoid dynamic type management. Dynamic type management means that type information can be added and removed from compiled programs at runtime. Achieving dynamic type management in C# is difficult, error prone, and not directly supported. The usage of Descriptors allows Whiskey2D to prevent loading and editing type information in program memory. We explored C#'s native dynamic type management, but we decided against using it. This was the central cause of a large Whiskey2D refactor, and thus will be discussed in greater detail in Section 6.1.1.

There are two primary types of Descriptors used in Whiskey2D's Editor, FileDescriptors, and InstanceDescriptors. FileDescriptors are used to model information that will eventually be stored in a file. There are various subclasses of FileDescriptor, including CodeDescriptors, and LevelDescriptors. InstanceDescriptors, on the other hand, model instantiations of designers' GameObjects. All of the Descriptor subclasses can be seen in Figure 3.10.

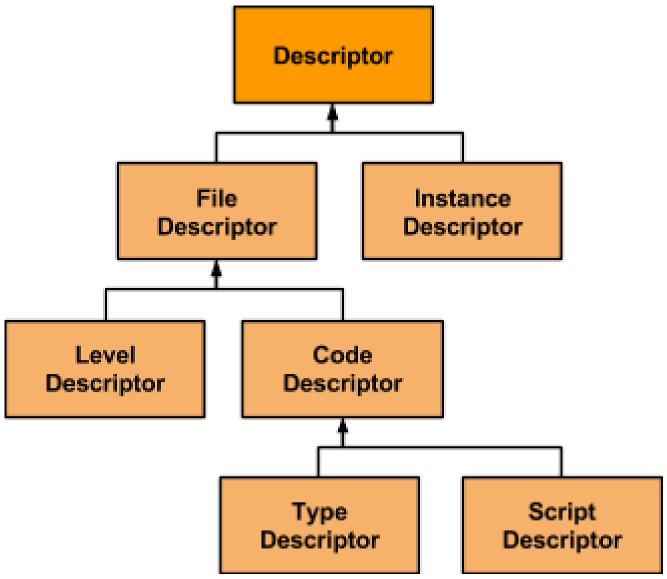


FIGURE 3.10. All of the Descriptor subclasses

FileDescriptor and the FileManager

FileDescriptors are Descriptors that model anything that will be generated to a file. The two main classes of FileDescriptors are CodeDescriptors and LevelDescriptors. Each of them generates data in a file that will be used later by Whiskey2D. All FileDescriptors are managed by a singleton class called the FileManager. The FileManager is responsible for keeping the Project's FileDescriptors in sync with the *.gamedata* file.

CodeDescriptor

One kind of FileDescriptor is a CodeDescriptor. The CodeDescriptor keeps track of C# code information, such as *class name*, *namespace* and *using statements*. When the CodeDescriptor is asked to generate its model, it generates C# source code. The Editor uses two kinds of CodeDescriptors: TypeDescriptors and ScriptDescriptors.

A TypeDescriptor models a subclass of GameObject. As discussed in Chapter 2, a GameObject has a set of properties that every subclass will inherit. A TypeDescriptor keeps track of default values for these properties, as well as any additional properties of the subclass. TypeDescriptors can generate valid source code for the type they are modelling.

Similarly to TypeDescriptor, ScriptDescriptor models a subclass of Script. Every Script must implement *onStart(), onUpdate(),* and *onClose().* A ScriptDescriptor stores the code that will be run in each method, and can use it to generate valid Script source code. The source code is then used to build the full Whiskey2D game and the Descriptor is discarded.

Level Descriptor

The second primary type of FileDescriptor is the LevelDescriptor. The LevelDescriptor doesn't generate human readable source code like CodeDescriptors do, but instead generate the actual GameLevel data that Whiskey2D will use to run a Level. A LevelDescriptor contains all of the data needed for a Level, with the exception of the Level's GameObjects. The designer's GameObjects will not be compiled until the build process, making it impossible for the LevelDescriptor to use them. Instead, the LevelDescriptor contains a set of InstanceDescriptors.

InstanceDescriptor

InstanceDescriptors are the last kind of major Descriptor used in Whiskey2D. An InstanceDescriptor describes an instance of a GameObject. Similar to how an object should be created with a defined type in C#, all InstanceDescriptors must be created with an existing TypeDescriptor. The TypeDescriptor gives the InstanceDescriptor all of its properties and initial configurations. For example, if the designer wanted to have a GameObject called Bike in their game, then they would need a TypeDescriptor called BikeType, and an InstanceDescriptor created with BikeType. Figure 3.11 below shows how TypeDescriptors and InstanceDescriptors map to real GameObjects in a compiled game.

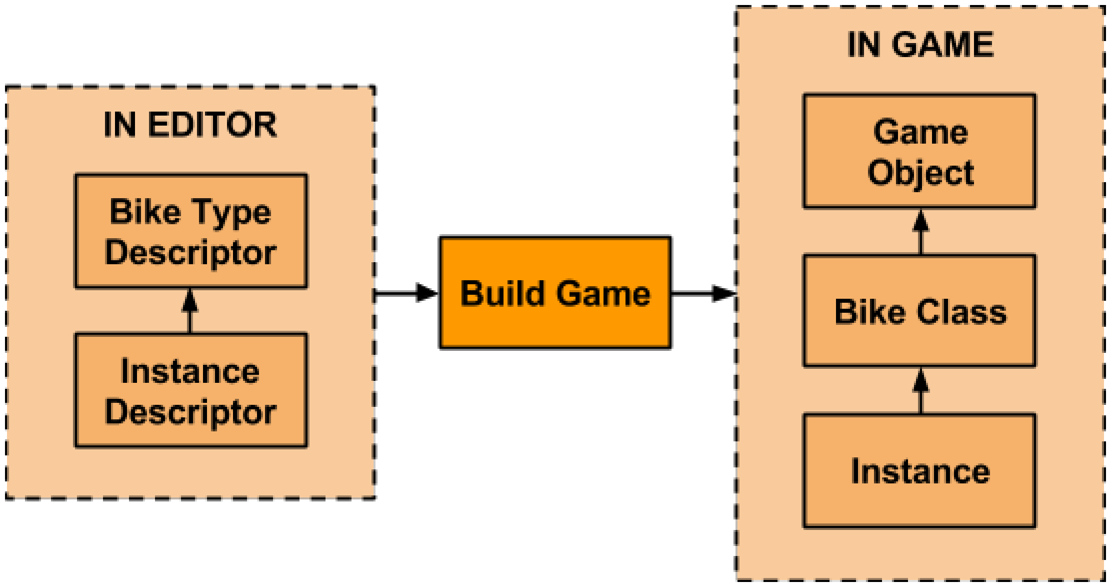


FIGURE 3.11. This process diagram shows in-editor objects being converted into their in-game representation.

Again, the advantage to using Descriptors is that they can undergo changes without invoking any sort of compilation. In the example above, if the designer wanted to modify the BikeType so that it had an integer property, *WheelRadius*, they can simply modify the TypeDescriptor. The edits to the TypeDescriptor will immediately trigger updates in all InstanceDescriptors using the modified TypeDescriptor. The result of adding a property to a TypeDescriptor will be that all of the associated InstanceDescriptors will gain the property. The clearest way to think of TypeDescriptors and InstanceDescriptors is to picture TypeDescriptor as a C# class, and InstanceDescriptor as an instance of the C# class. Figure 3.12 shows how all Descriptors map to their real types and objects in a compiled game.

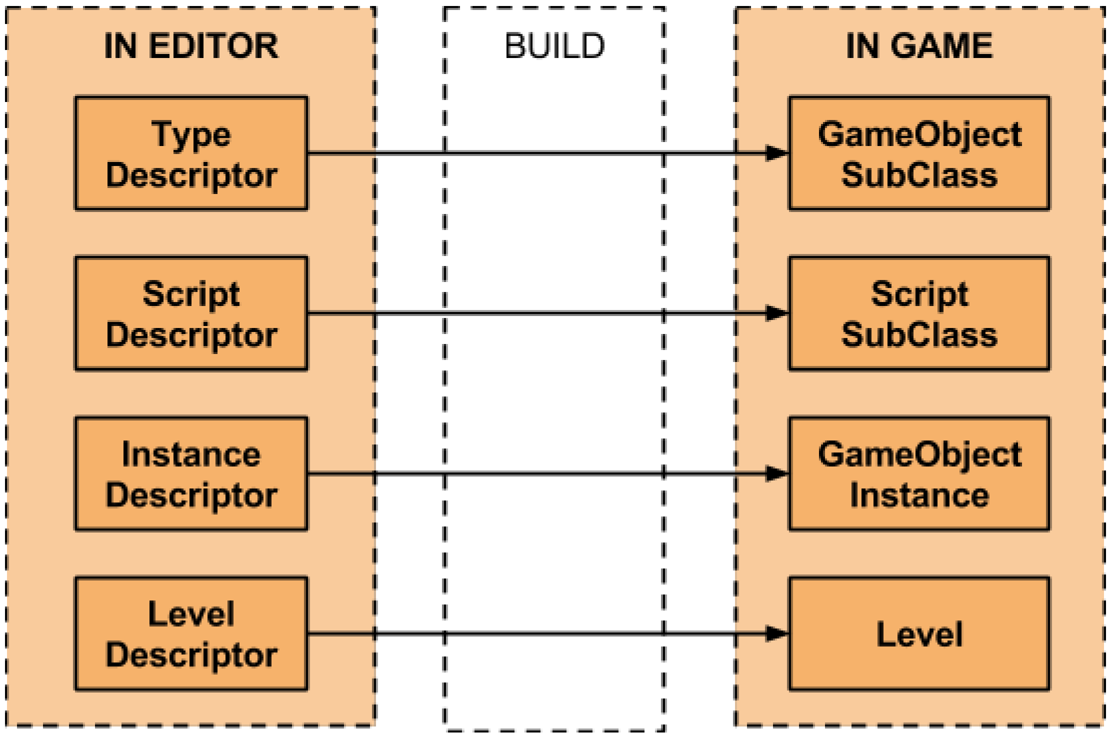
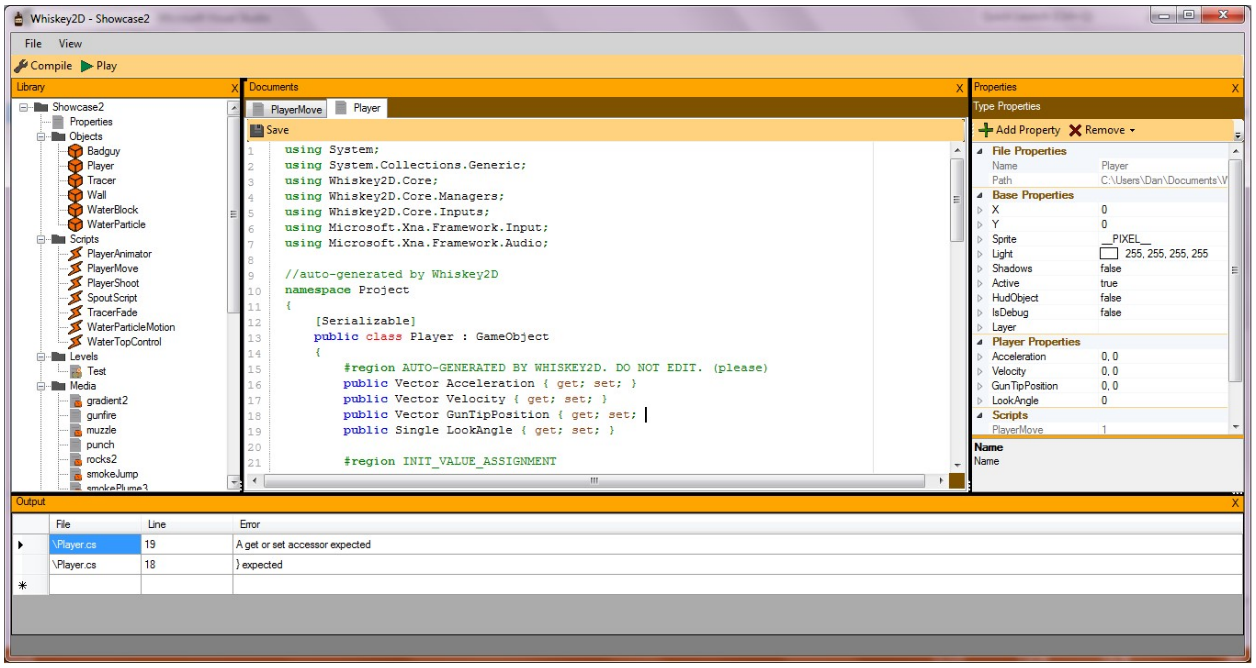


FIGURE 3.12. This process diagram shows in-editor Descriptors being converted into their in-game representation.

3.2.3 User Interface

The last primary subsystem that the Editor provides is the User-Interface (UI) itself. The UI gives the designer a visual way to interact with and construct their games. Whiskey2D's Editor is written using WinForms. With the exception of a few utility windows, all of the Editor's frames and views were built manually - i.e., not using the Visual Studio WinForm builder.

Figure 3.13 shows the Editor UI. The UI is made up of three primary views, with a few other views providing support. The Library View allows the designer to interact with their game files. The Document View lets the designer edit the core content of a file, and the Property View gives additional control. A critical example of a Document View is the WhiskeyControl view. The WhiskeyControl view is what renders Levels as they are being built.



```
1   using System;
2   using System.Collections.Generic;
3   using Whiskey2D.Core;
4   using Whiskey2D.Core.Managers;
5   using Whiskey2D.Core.Inputs;
6   using Microsoft.Xna.Framework.Input;
7   using Microsoft.Xna.Framework.Audio;
8
9   //auto-generated by Whiskey2D
10  namespace Project
11  {
12      [Serializable]
13      public class Player : GameObject
14      {
15          #region AUTO-GENERATED BY WHISKEY2D. DO NOT EDIT. (please)
16          public Vector Acceleration { get; set; }
17          public Vector Velocity { get; set; }
18          public Vector GunTipPosition { get; set;
19          public Single LookAngle { get; set; }
20
21          #region INIT_VALUE_ASSIGNMENT
```

File Properties	
Name	Player
Path	C:\Users\Dan\Documents\W
Base Properties	
X	0
Y	0
Sprite	__PIXEL__
Light	▢ 255, 255, 255, 255
Shadows	false
Active	true
HudObject	false
IsDebug	false
Layer	
Player Properties	
Acceleration	0, 0
Velocity	0, 0
GunTipPosition	0, 0
LookAngle	0
Scripts	
PlayerMove	1

Name
Name

	File	Line	Error
▶	\Player.cs	19	A get or set accessor expected
	\Player.cs	18	} expected
*			

FIGURE 3.13. A screenshot of Whiskey2D's editor GUI, with the Library on the left, Properties on the right, the Documents in the center, and Output on the bottom.

Library View

The first primary view of the Editor is the Library View. It displays all of the files associated with a Project, including all GameObject Types, Scripts, Art assets, Levels, and a Project properties file. As mentioned in Chapter 2, the designer can use this view to add, remove, or open files. Figure 3.14 is a zoomed in image of the Library view.

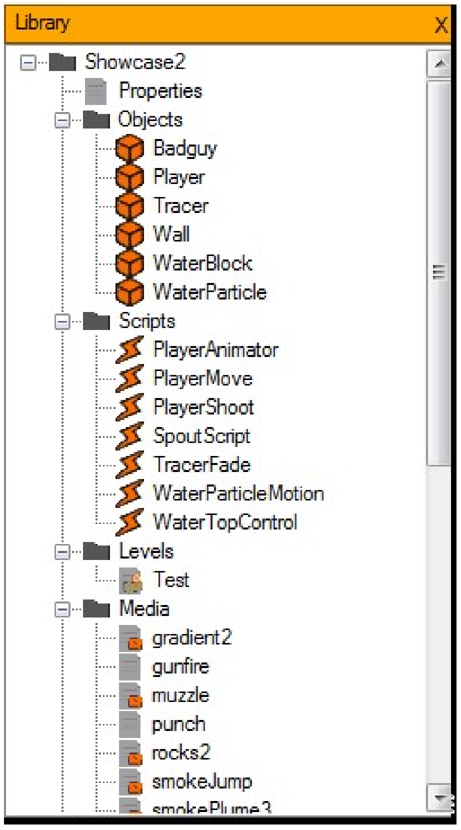
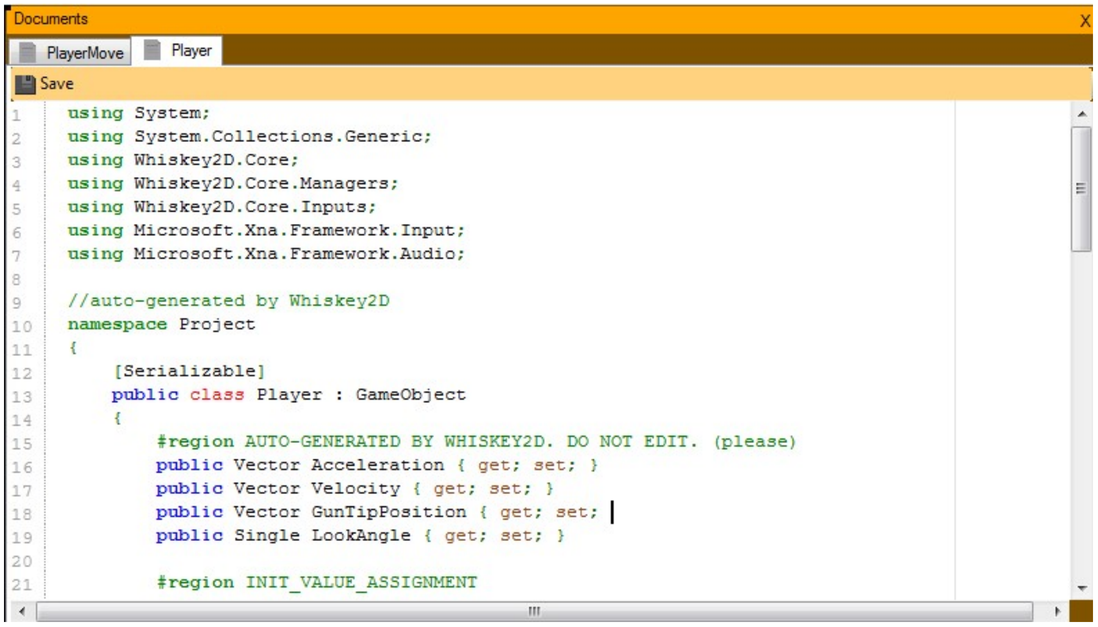


FIGURE 3.14. The Library view, which shows all of the files used in the active project.

Documents and Properties

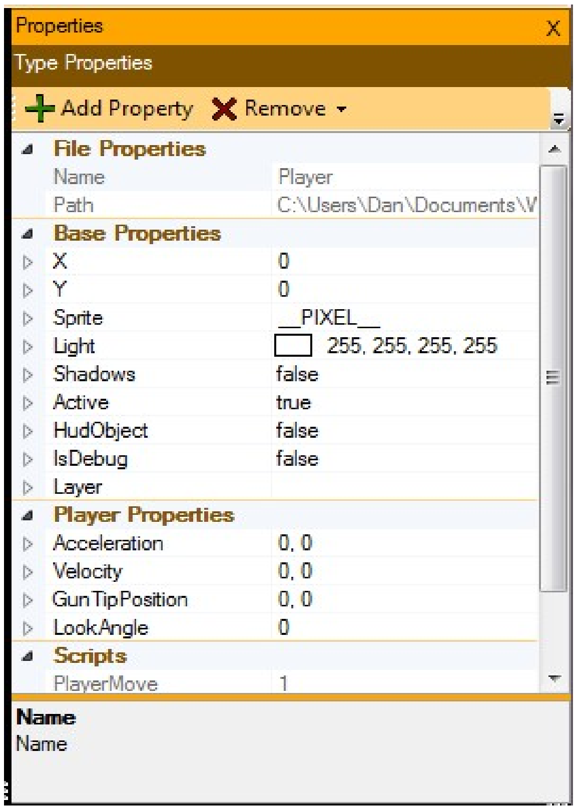
When a designer opens a file from the Library, the Document View and Property View change to reflect the new file. Opening different kinds of files will result in different types of views being opened for the designer. For example, Figure 3.15 shows the Document view as it looks when it is associated with a GameObject. This can also display

Levels, art assets, Script code, and more. Figure 3.16 shows the Property view when looking at the Player type.



```
using System;
using System.Collections.Generic;
using Whiskey2D.Core;
using Whiskey2D.Core.Managers;
using Whiskey2D.Core.Inputs;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Audio;

//auto-generated by Whiskey2D
namespace Project
{
    [Serializable]
    public class Player : GameObject
    {
        #region AUTO-GENERATED BY WHISKEY2D. DO NOT EDIT. (please)
        public Vector Acceleration { get; set; }
        public Vector Velocity { get; set; }
        public Vector GunTipPosition { get; set; |
        public Single LookAngle { get; set; }

        #region INIT_VALUE_ASSIGNMENT
```

FIGURE 3.15. The Document view, currently showing GameObject code for the Player type.

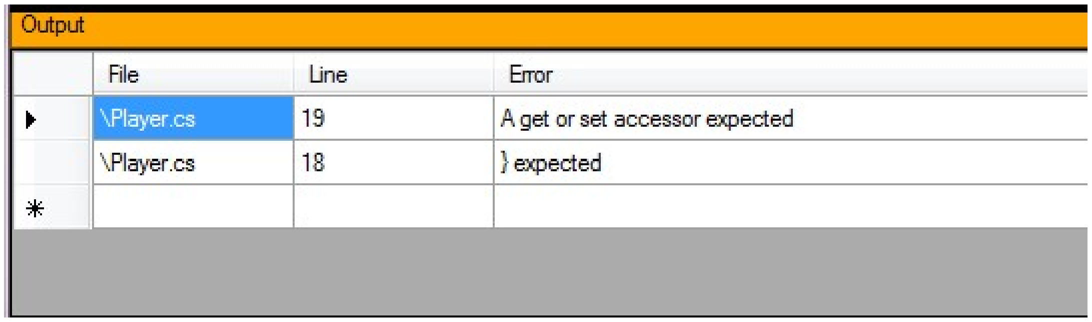


File Properties	
Name	Player
Path	C:\Users\Dan\Documents\V
Base Properties	
X	0
Y	0
Sprite	__PIXEL__
Light	⬜ 255, 255, 255, 255
Shadows	false
Active	true
HudObject	false
IsDebug	false
Layer	
Player Properties	
Acceleration	0, 0
Velocity	0, 0
GunTipPosition	0, 0
LookAngle	0
Scripts	
PlayerMove	1

Name
Name

FIGURE 3.16. Property view currently showing the properties for the Player type.

Whiskey2D also has an Output view, which can be seen at the bottom of the editor UI. Figure 3.17 below shows the Output view for the same project. Here, the designer forgot a '}', and the output is telling them that.



	File	Line	Error
▶	\Player.cs	19	A get or set accessor expected
	\Player.cs	18	} expected
*			

FIGURE 3.17. The Output view in Whiskey2D's editor UI.

The process of generating the appropriate WinForm views for different kinds of files is handled by a utility class called the *DocumentContentFactory*. The DocumentContentFactory is responsible for creating both the Document View, and the Property View for a given Descriptor. Recall from the previous Section on Descriptors, that essentially every important piece of data in the Editor is a Descriptor. The DocumentContentFactory takes a Descriptor, and uses it to output the appropriate views. This system is robust, because it allows new types of Descriptor to be added without a lot of work.

WhiskeyControl

One extremely important view is the WhiskeyControl. The WhiskeyControl is responsible for rendering a designer's Level while they are developing it. The WhiskeyControl is also in charge of handling any input that directly controls

InstanceDescriptors in a Level. The WhiskeyControl uses several utilities from the Core so that it can model an actual Whiskey game as closely as possible. The WhiskeyControl manages a game-preview in WinForms, made solely out of Descriptors, whereas MonoGame manages an actual game built out of compiled GameObjects and Scripts.

Since Instance Descriptors are specialized Game Objects, they can be rendered and updated by the Core. The WhiskeyControl simulates a lightweight version of the Core's Game Loop in order to manage all Instance Descriptors. Each Instance Descriptor can be selected, scaled, rotated, moved, and deleted. All of these actions are performed through customized Scripts. A specialized GameObject, called *ObjectController* exists in WhiskeyControl. The ObjectController has several attached Scripts that facilitate the Instance transformations.

All of the Input that the ObjectController's Scripts need comes through the same Input system described in the previous Section. However, a new Input source, called the EditorInputSource, is used. It captures data from the WinForms architecture, and converts into a format that Whiskey Scripts can interpret.

Finally, the WhiskeyControl overrides most of the Core's default rendering technique. The Core's rendering, and the Editor's rendering are quite similar, but differ in key ways, so as to warrant specialized code in the Editor. For example, the Editor's rendering must take into account the grid system in the background of a Level, the selection sprites surrounding the selected GameObject, as seen in Figure 3.18, and must allow users to customize various rendering features in real time.

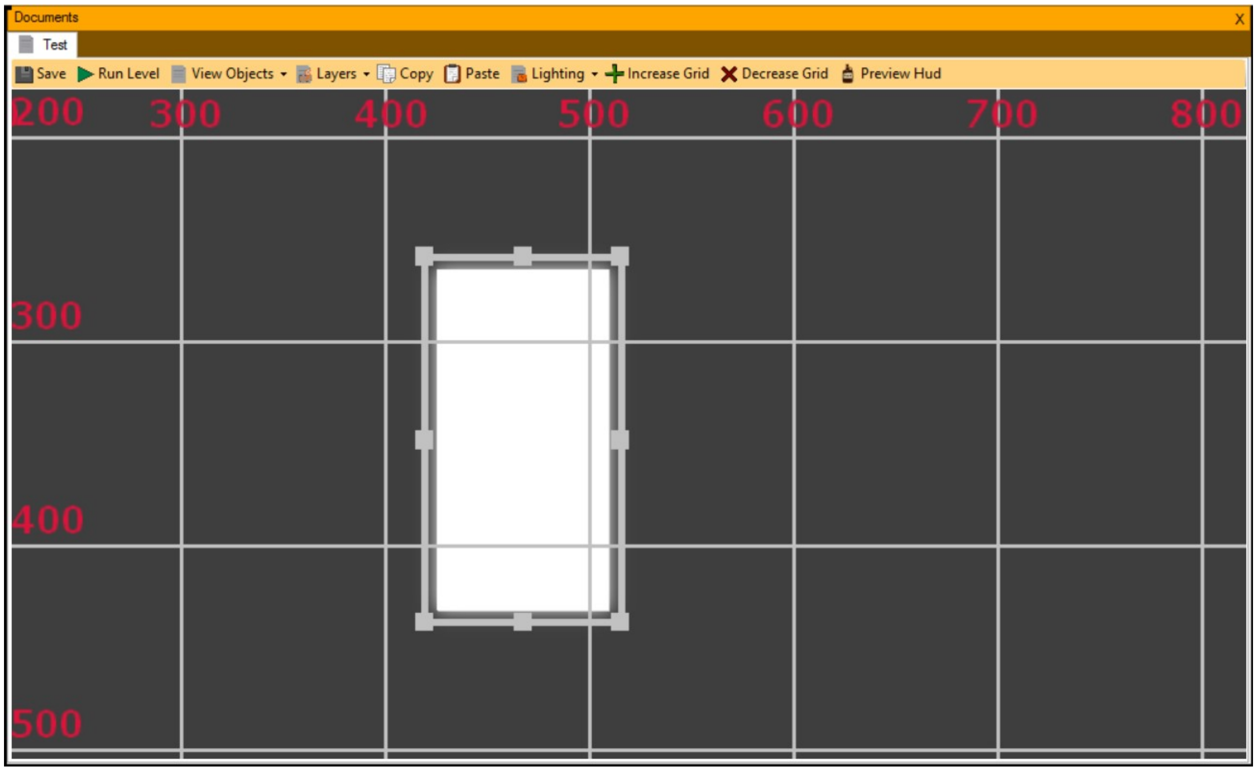


FIGURE 3.18. A close up of the WhiseyControl, displaying gridlines, and a selection box around the sprite

The other vital difference between the Core and the WhiskeyControl is the way they actually draw to the screen. The Core uses MonoGame directly to draw content to a window. The WhiskeyControl cannot generate a separate window for each Level, and so it has to secure a Graphics pane to render to inside of WinForms.

<u>3.3 Game Data Life Cycle</u>

Every game made with Whiskey2D goes through the same data stages. While the designer is developing the game, most of the data exists inside the .gamedata file in the project directory. When the designer builds or tests the game, the data in the .gamedata file is compiled into a usable DLL, which is then loaded and executed by an executable called Launcher. Figure 3.19 below visualizes this creation.

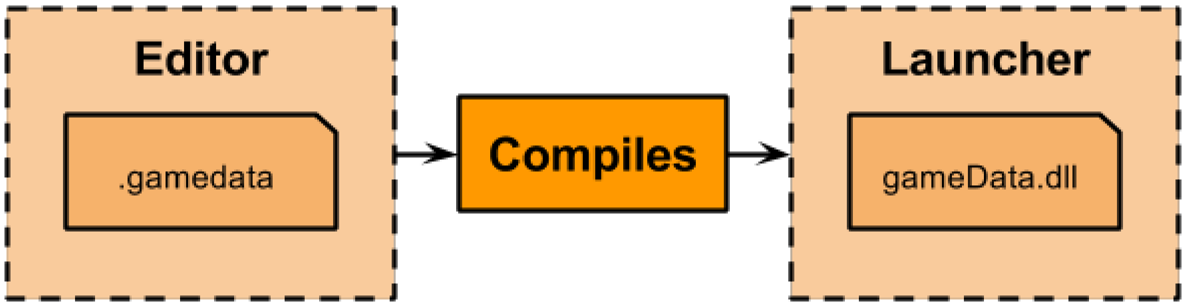


FIGURE 3.19. A process diagram, showing that the .game file becomes the gameData.dll

3.3.1 Editor Serializations

The first stage of a game's data life is the .gamedata file. The .gamedata file is a binary formatted serialization of the Editor's FileManager. Every FileDescriptor that is being tracked by the FileManager will be serialized and stored inside the .gamedata file. When the Editor loads a project, it deserializes the data from the .gamedata file, and uses it to repopulate the FileManager.

3.3.2 GameData.dll

The next state of a game's data cycle is the gameData DLL file. When the designer builds the game, the contents of the .gamedata are deserialized, and the resulting CodeDescriptors are used to generate source code. The Editor then compiles all of the generated source code into a single DLL, this is the gameData DLL.

The DLL contains actual compiled type information that the CodeDescriptors were modelling. The type information can be loaded into the Editor, and used to convert each InstanceDescriptor in every Level into a usable GameObject. Each new GameObject will be of the appropriate type, and will have all of the properties of the InstanceDescriptor. Each Level's InstanceDescriptors will undergo the conversion, and the resultant GameObjects will be serialized into GameLevel files. In Figure 3.20, you can see Instance Descriptors that exist in the editor being converted to GameObject instances in the game.

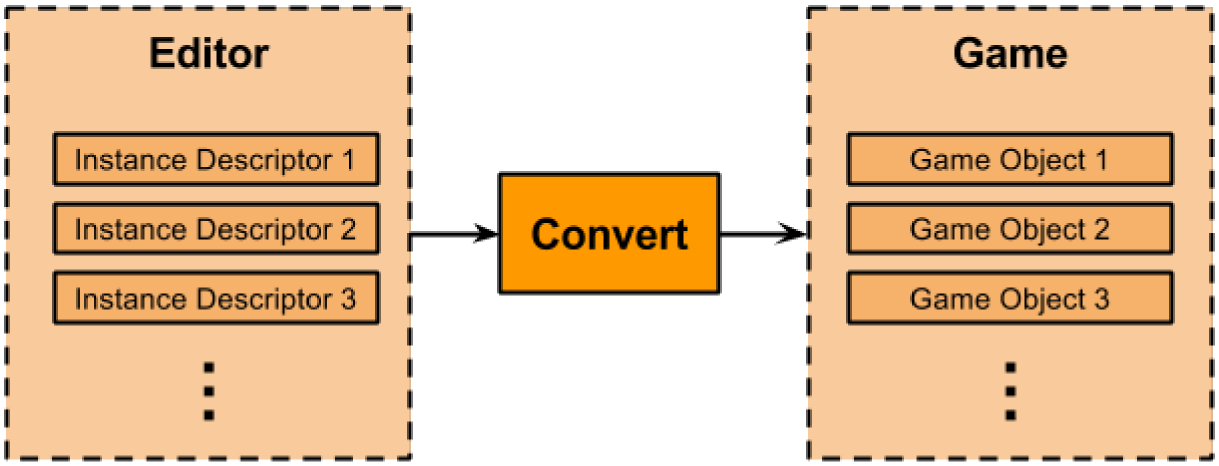


FIGURE 3.20. A process diagram showing that instance descriptors are mapped to GameObject instantiations

The last stage of the compilation process is to build the output game directory in full. A subdirectory called *build* is created inside the project directory. All of the required library files, including gameData.dll are put in the build directory, along with all the necessary media files to run the game. The level files and a configuration file are all put in the *build* directory as well. Finally, a copy of Launcher.exe is created, renamed to the game's title, and copied into the *build* directory.

3.3.3 Launcher.exe & Target Platforms

Every game made with Whiskey2D will use the Launcher.exe utility. The executable is programmed to search for a local DLL called gameData.dll, and load its types into memory. Then the program will search for a configuration file that tells it what level to load from the local *levels* directory. Launcher.exe is nothing but a host executable for the Core to take over and start the Game Loop with the designer’s data. The executable immediately passes all control to the Core. Using a standard executable like Launcher.exe helps strengthen the idea that a designer’s game is nothing more than media, levels, and a gameData.dll. All of the backend code is always the same, including the actual .exe itself.

Chapter 4 - Example Game: Mechanical Plunge

In order to demonstrate the capabilities of Whiskey2D, we created a side-scrolling platformer game called Mechanical Plunge (Figure 4.1). The game was created in the final stages of Whiskey2D's development to prove that the software could be used effectively to create non-trivial games. We partnered with Pat Gallagher and Bryce Rabideau for artistic and musical assistance.

FIGURE 4.1. The logo and title for Mechanical Plunge

Mechanical Plunge is a shooter-platformer game. The game follows a robot, who is trying to escape an abandoned science laboratory, located deep underground. Throughout the game, the robot must overcome jump puzzles, battle cave dwelling monsters, and solve simple riddles to open new paths forward.

To play Mechanical Plunge, as seen in Figure 4.2, launch the MechanicalPlunge.exe inside the *samples/showcase2/build* directory of the Git repository. The character moves with the W, A, and D keys, and aims and shoots with the mouse. Pressing A or D will move the character sideways, and pressing W, will cause the character to jump. Moving the mouse will change the character's aim, as denoted by a red reticle. Pressing the Left Mouse button will make the character shoot his pistol. The objective of the game is escaping the cave. To do this, you must open the two blast doors located at the end of the level.

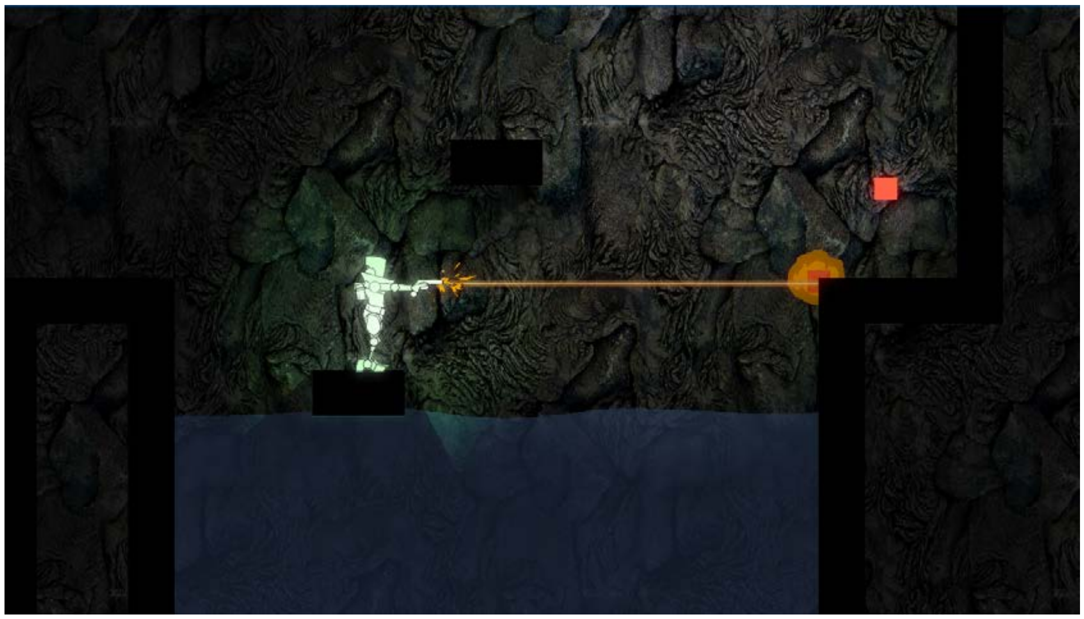


FIGURE 4.2. A screenshot from Mechanical Plunge. The player can be seen shooting towards the right.

Our primary goal in creating Mechanical Plunge was for it to showcase the capabilities of Whiskey2D. As such, we built the game to take advantage of as many Whiskey2D features as possible. Notably, we made use of the built in collision, camera, and rendering systems. We also used several of the shared GameObjects, provided with every Whiskey2D project.

The character in Mechanical Plunge is a culmination of the collision system, and the sprite sheet animation system. The character is controlled by a simple bounding box that is moved around the screen by a PlayerMovement Script. The Script uses Whiskey2D's collision system to collide with the level geometry, and keep the player from passing through objects. Some of the collision code can be seen in the code snippet (Figure 4.3).

```
Collisions<Wall> wallColls = Gob.currentCollisions<Wall>(); // get all collisions with walls
bool surface = Vector.Zero;
foreach (Collision<Wall> c in wallColls) { // for all wall collisions

        Gob.Position -= c.MTV; //subtract the min translation vector

        // update the velocity, according the normal of the collision
        Gob.Velocity = c.Normal.Perpendicular * c.Normal.Perpendicular.dot(Gob.Velocity);
        surface = c.Normal;

        if (c.Normal.Y > .8f) { // is the player back on the ground?
                jumping = false;
        }
}
```

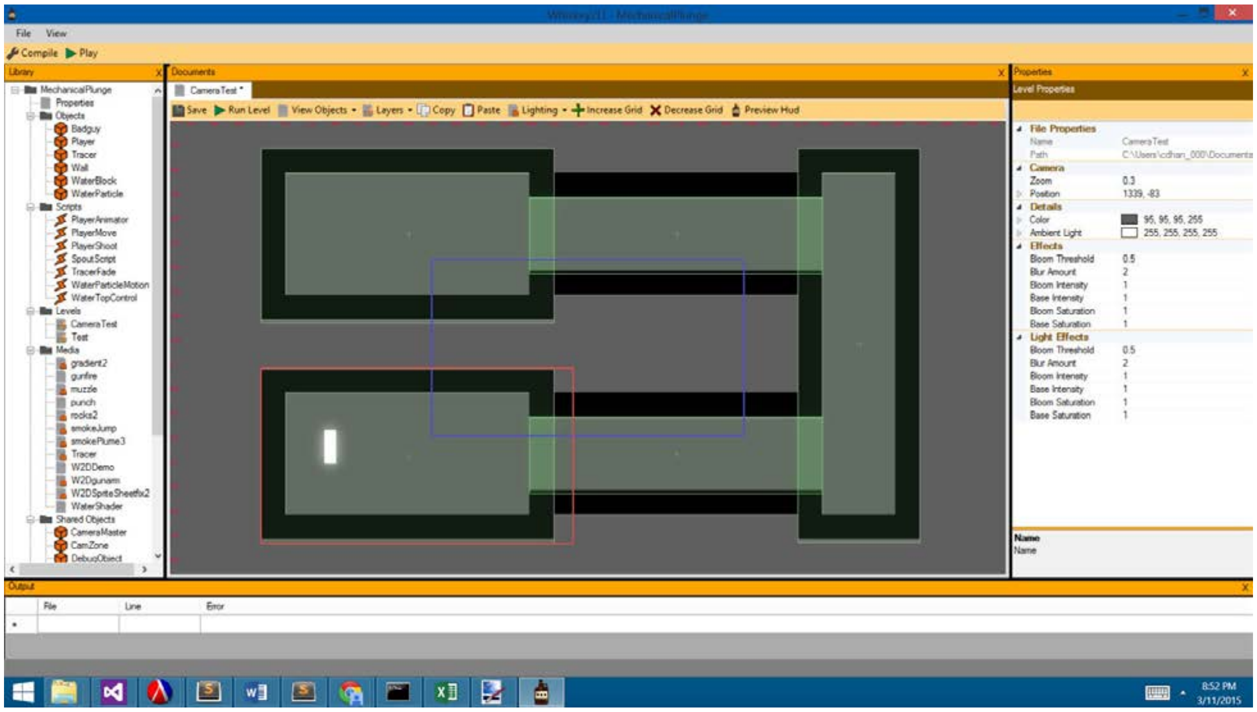
FIGURE 4.3. Code snippet showing how the collision system works

The bounding box is invisible, and the character's visuals come from the sprite sheet animation system. There is a second Script, PlayerAnimator, that updates the visual appearance of the character. PlayerAnimator observes the current acceleration and velocity values of the player to decide which animation segment to play. A snippet of the animation Script (Figure 4.4) can be seen below. Due to the changes in perceived player width during a standing pose, and a running pose, the PlayerMovement Script will change the size of the bounding box used for collisions, depending upon the acceleration and velocity of the player.

```
public override void onStart() {

       // initialize runLeft, and runRight from the sprite sheet
       runLeft = vis.Sprite.createAnimation(8, 15, 7, true);
       runRight = vis.Sprite.createAnimation(0, 7, 7, true);
}

public override void onUpdate() {

       if (Gob.Acceleration.X > 0 || Gob.Velocity.X > 2f) { // is the player going right?

              runRight.advanceFrame();
              runRight.Speed = 8 - (int) Math.Abs(Gob.Velocity.X) / 3;

       } else if (Gob.Acceleration.X < 0 || Gob.Velocity.X < -2f) { // or left?

              runLeft.advanceFrame();
              runLeft.Speed = 8 - (int)Math.Abs(Gob.Velocity.X)/3;
       }
}
```

FIGURE 4.4. Code snippet showing how animations may be used and created

In addition to the character movement and animation, the built-in Whiskey2D camera was set to track the character in interesting ways. Camera Zones were placed around the level that will specify how the camera tracks the player. Normally, the camera will center on the character. However, the camera is bound by the camera zones, so when the player approaches the edge of a camera zone, the character will be allowed to move out of center screen, as the camera cannot follow them. The placement of camera zones is quite simple, and can be seen in the figure (Figure 4.5) below.



File Properties	
Name	CameraTest
Path	C:\Users\cdhan_000\Documents
Camera	
Zoom	0.3
Position	1339, -83
Details	
Color	95, 95, 95, 255
Ambient Light	255, 255, 255, 255
Effects	
Bloom Threshold	0.5
Blur Amount	2
Bloom Intensity	1
Base Intensity	1
Bloom Saturation	1
Base Saturation	1
Light Effects	
Bloom Threshold	0.5
Blur Amount	2
Bloom Intensity	1
Base Intensity	1
Bloom Saturation	1
Base Saturation	1

Name
Name

FIGURE 4.5. A shot showing the placement of CamZones. The CamZones are the green boxes, stretched between the walls of the level.

Camera Zones are an example of a shared GameObject. Other shared GameObjects include Sound Zones, and SpriteEffects. A SoundZone can be configured to play sounds when the character is entering or exiting the zone. These are used in Mechanical Plunge to trigger the level's audio, as well as various environmental sounds specific to different level areas. The SpriteEffect is another shared GameObject, that is used to play an animated sprite sheet on the fly. This GameObject is used extensively in our game. Examples of the SpriteEffect in Mechanical Plunge are the muzzle flash of the character's pistol, the smoke plume when he jumps. The SpriteEffect is invoked from Script, and a can be seen in the code snippet (Figure 4.6) below.

```
if (Input.isNewKeyDown(Keys.W)) { // is the up key being pushed ?
       if (surface.Y > .8f) { // is the player on the ground?

              // set logic for jumping
              jumping = true;
              jumpCounter = 40;
              Gob.Acceleration -= gravity * 12;
              doubleJumped = false;

               //create jump effect
              SpriteEffect fx = new SpriteEffect(Level);
              fx.Position = Gob.Position + Vector.UnityY*Gob.Bounds.Size.Y /4;
              fx.Effect = "smokeJump";
              fx.Frames = Vector.One * 4;
              fx.Sprite.Scale *= .3f;
              fx.Speed = 4;
       }
}
```

FIGURE 4.6. A code snippet showing a SpriteEffect might be used for a jump effect

Another strong feature of Whiskey2D is its ability to allow for custom rendering. As mentioned in previous Chapters, each GameObject can override its rendering behavior, on both the image pass, and the lighting pass. In Mechanical Plunge, the laser effect that occurs when the character shoots takes advantage of custom rendering. The laser overrides the lighting pass to render its image onto the lighting mask, which creates an elongated light emitter, instead of the default gradient light image.

The other way that Whiskey2D supports custom rendering is its layering system. Each layer can provide its own shader, and post-shader. The water in Mechanical Plunge is on its own layer, and has a post-shader running that makes the water droplets blend together. The water is also overriding the default rendering behavior, and using primitives to draw, instead of sprites. The water's rendering components can be seen in the figure (Figure 4.7) below.

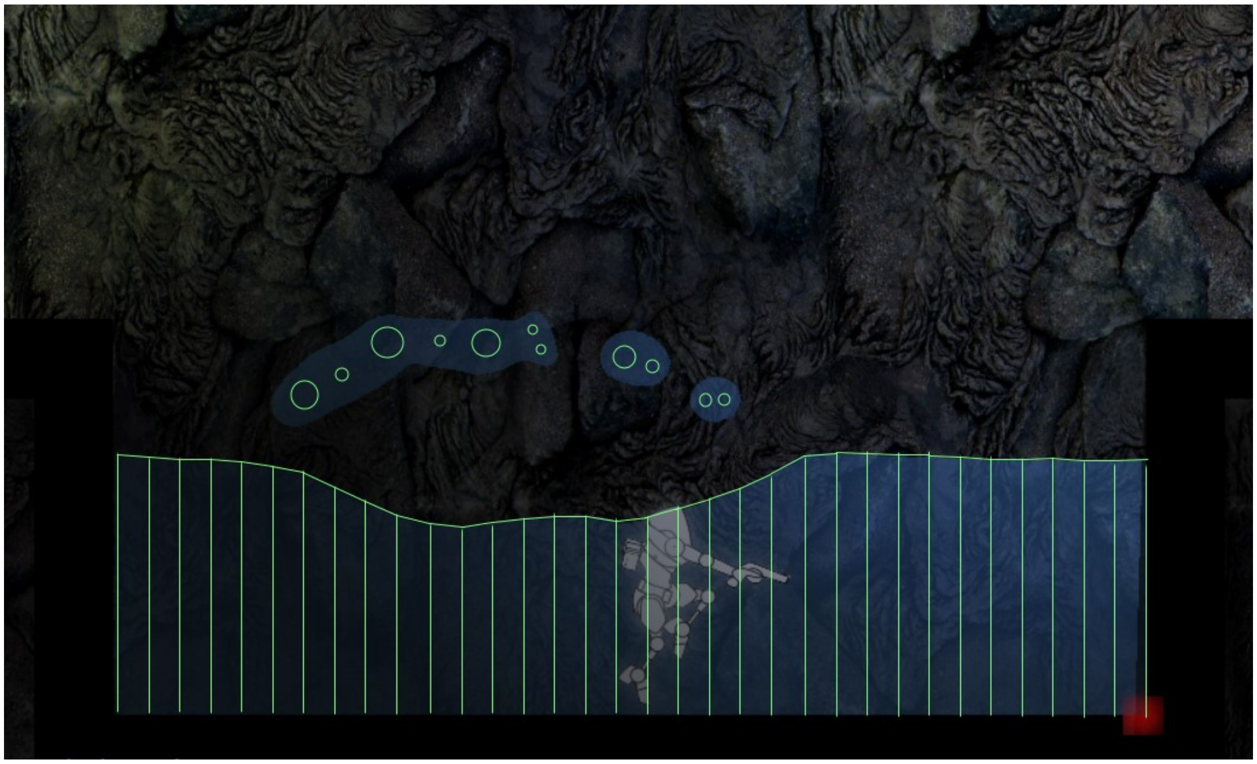


FIGURE 4.7. A screenshot of the water in Mechanical Plunge. The body of water is drawn as quadrilateral primitives, and the water droplets are drawn as metaballs.

Chapter 5 - Developing Whiskey2D

Building Whiskey2D was a challenge. We adhered to a disciplined software engineering process that enabled us quickly build powerful code. Throughout our entire development time, from late July to early March, we built our code iteratively and through rapid prototyping. We used a wealth of tools to support our team, mainly including Visual Studio and Google Docs. At key points in Whiskey2D's development, we paused to hold usability testing sessions. In this Chapter, we will discuss the history of Whiskey2D, our team's style, and how we conducted usability tests.

5.1 Development History

Officially, we began our project on August 28th, the first day of A-term. However, we had been planning the initial designs of Whiskey2D since late July. Over the next 30 weeks, we brought Whiskey2D from a blank project, to a near-professional quality piece of software. Whiskey2D underwent multiple of improvements, revisions, and refactorizations along the way. Figure 5.1 gives a visual representation of when we were working on certain aspects of the project and for how long.

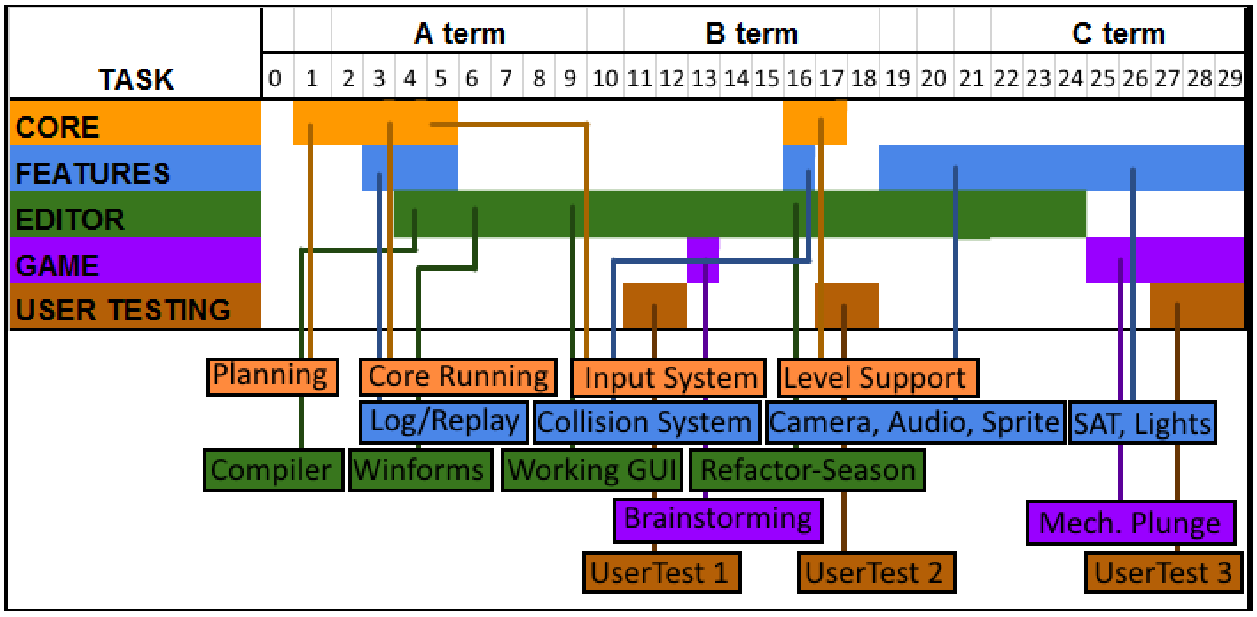


FIGURE 5.1. This is a Gantt chart, showing where our efforts were spent each week of development.

5.1.1 Timeline and Narrative

The following is a chronological development history of our major checkpoints and progress indicators, taken from our notes at the time, our meeting agendas, and our Git repository.

7 / 27 / 2014 (proposal created) - 9 / 9 / 2014 (first meeting)
Initial Planning

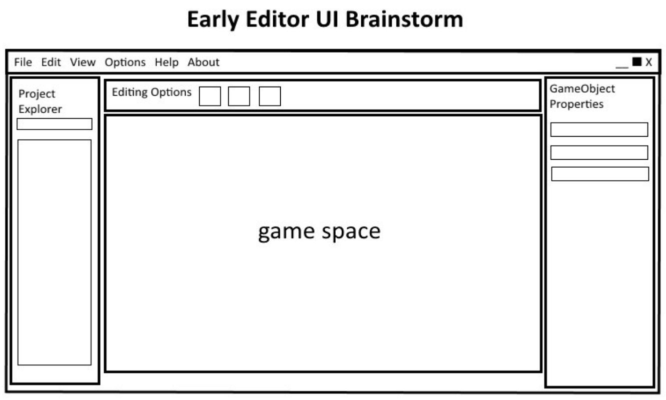


FIGURE 5.2. Our first concept sketch for the UI of Whiskey2D

As summer comes to an end we meet a few times to begin writing up and finalizing the proposal for what later became Whiskey2D. After receiving word that our MQP proposal was accepted, we begin researching other similar game engine products on the market. At this time we also start designing concepts for our UI as well as the structure for our engine (Figure 5.2).

At this time we also start documenting an initial class diagram and class structure for Whiskey2D's Core.

9 / 13 / 2014

Whiskey Core

Whiskey2D's Core is up and running! The engine is able to run simple games which we created as a proof of concept (see in Figure 4.3). Our initial games are built with the very same philosophy as Whiskey2D currently employs. The current system of GameObjects and their Scripts controlling each of their behaviors are present. The Core contains most of the Managers that currently exist in Whiskey2D, and work functionally the same.

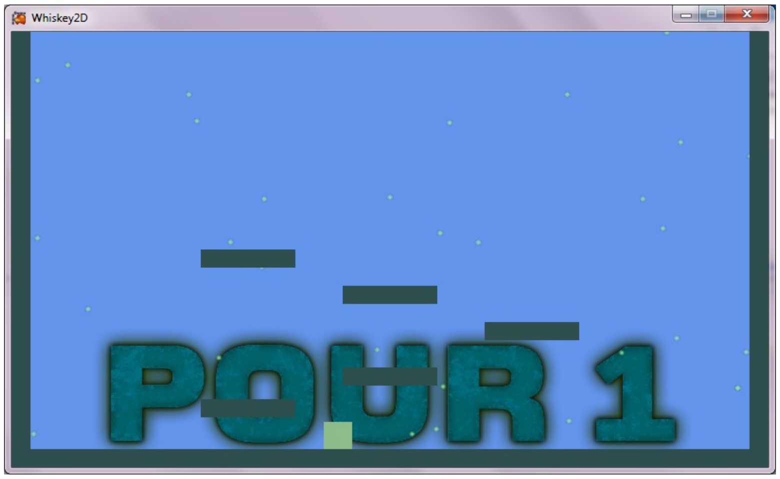


FIGURE 5.3. The first game ever created with the Whiskey2D core.

Sample games are able to process keyboard input from a user. We employed a rudimentary system for collision detection. Most of the games were side scrolling platformers, where all shapes were squares and circles.

9 / 15 / 2014

Rudimentary GUI for Compilation

An initial GUI is created in WinForms to facilitate the compilation of a game's data, seen in Figure 5.4. The GUI is able to facilitate creation of new GameObject type files and Script files. The GUI does not provide any way to edit files. In order to create interesting GameObject types or Scripts, the files can be edited in an external text editor. The compiler GUI can then compile the source code files to individual DLLs. There is no way to run these DLLs in a game at this point from the GUI, but this paves the way for future compilations of full Whiskey2D games.

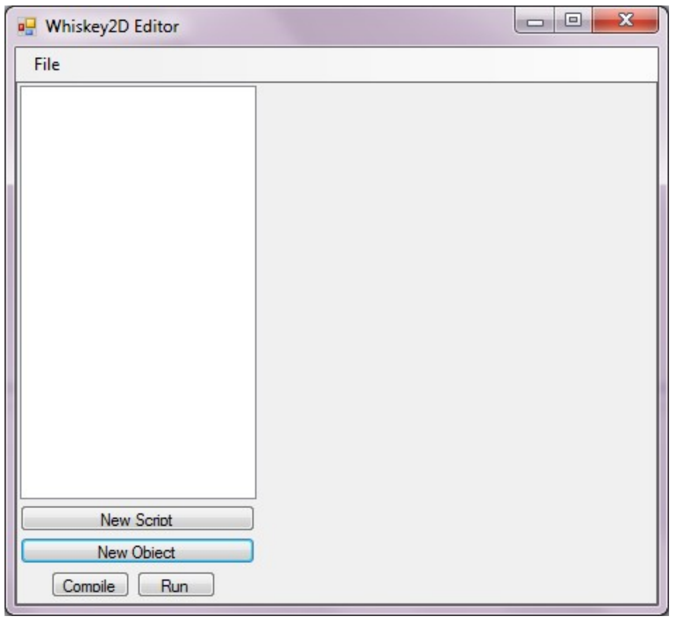
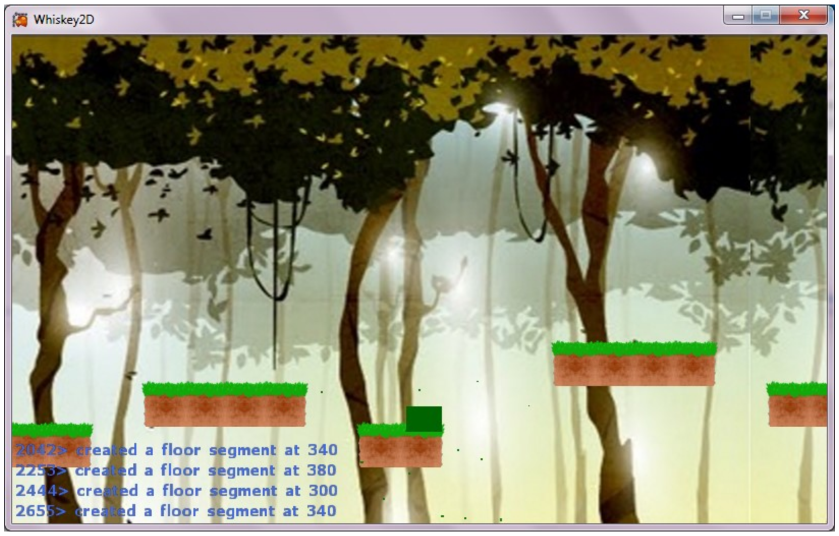


FIGURE 5.4. The first UI that would compile source code to class files.

9 / 20 / 2014

Log and Replay Feature

Log and replay system is created, which allows Users and designers to replay Sections of a level from a log file. Input sources are abstracted which allow input from sources other than the keyboard, including the mouse. A game is created that uses Log and Replay, as seen in Figure 5.5.



```
2042> created a floor segment at 340
2253> created a floor segment at 380
2444> created a floor segment at 300
2655> created a floor segment at 340
```

FIGURE 5.5. A game created that exemplified log and replay

9 / 23 / 2014

Monogame and WinForm Integration

A Monogame control is created, which allows Monogame to be inside WinForms. This can be seen in Figure 5.6. This is a first step to getting rendering and level editing into our UI, and the start of our GUI becoming an editor.

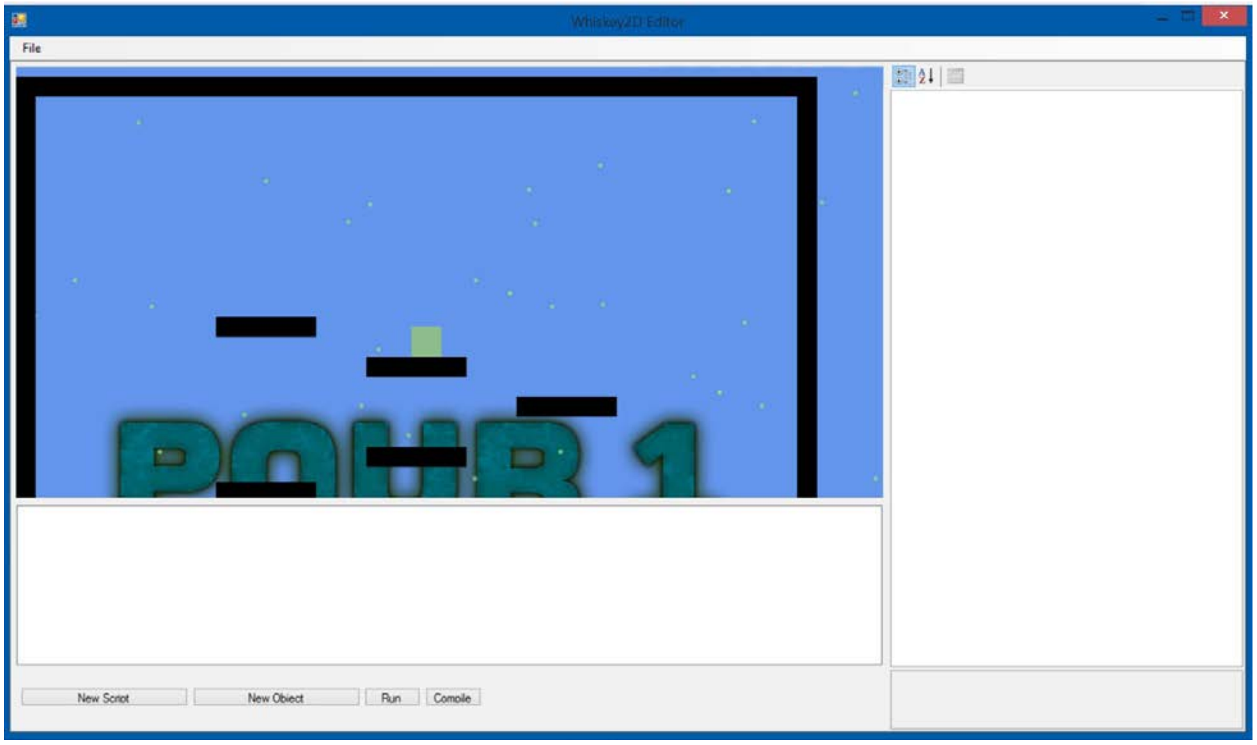


Figure 5.6. Screenshot of the first time the MonoGame Graphics could be rendered inside WinForms.

10 / 15 / 2014

First Full GUI

The first version of the Editor that can create a game level from start to finish is created (see Figure 5.7). This was the first time we were able to create a GameObject type, instance of it in a Level, a Script, and attach the Script to the GameObject. We were also able to build the Level, and play the resultant game. Several bugs exist in this editor, but it is a major milestone for the end of A term, and serves as a strong prototype for later versions.

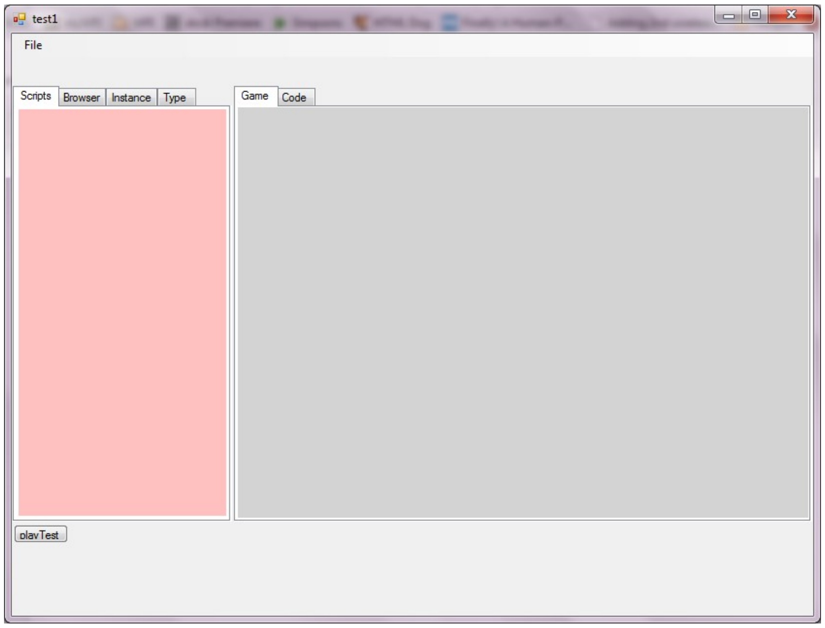


FIGURE 5.7. A screenshot of the UI as it stood at the time.

11 / 9 / 2014

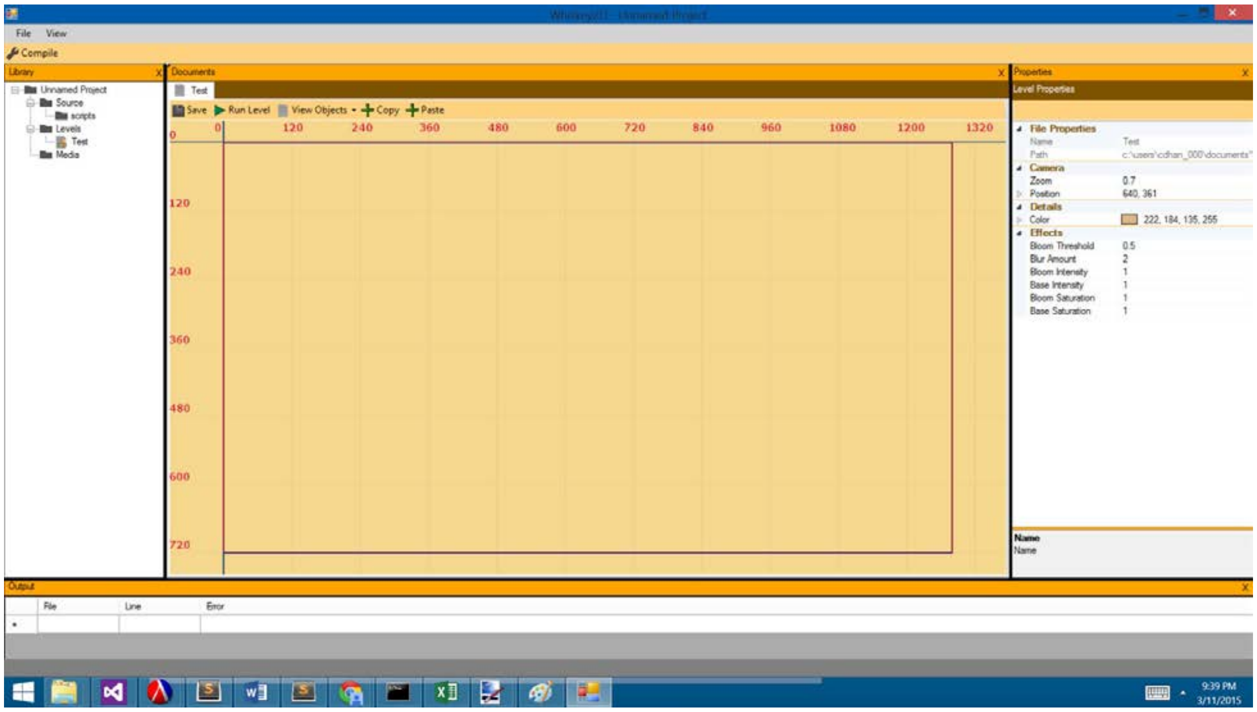
Usability Testing Round 1

We conduct our first round of usability testing with the current Editor. The users are not able to create a game, due to the weakness and instability of the Editor. This eventually leads us to understand that a major front-end and back-end refactor is required. (See Section 5.4 for details.)

11 / 15 / 2014 -- 12 / 16 / 2014

Refactored GUI

The large refactor that we started on 11/15, after the first usability test, is now complete. The Editor has been completely redone, both on the back-end and the front-end. The new front end can be seen in Figure 5.8. There is no longer any need for Dynamic Type management, described in Chapter 4 and 6, and the UI is professional and stable.



File Properties	
Name	Test
Path	c:\users\cdhan_000\documents"
Camera	
Zoom	0.7
Position	640, 361
Details	
Color	222, 184, 135, 255
Effects	
Bloom Threshold	0.5
Blur Amount	2
Bloom Intensity	1
Base Intensity	1
Bloom Saturation	1
Base Saturation	1

Figure 5.8. The UI after a long refactor

12 / 11 / 2014

Usability Testing Round 2

We conduct our second round of usability testing with a completely reworked piece of software. The users are able to create the shell of a game. User performance on this round of testing has vastly improved from the previous test, due to the new Editor. (See Section 5.4 for details.)

1 / 4 / 2015

Improved Media Support

Media support for Whiskey2D is greatly improved. Sprites can now support animations and Sounds can be loaded and played. Our first animated Sprite can be seen in Figure 5.9. In addition, a Level is no longer bound to exist in screen coordinates. Camera support is added so that Level size can be near arbitrary, bounded only by floating point limitations.

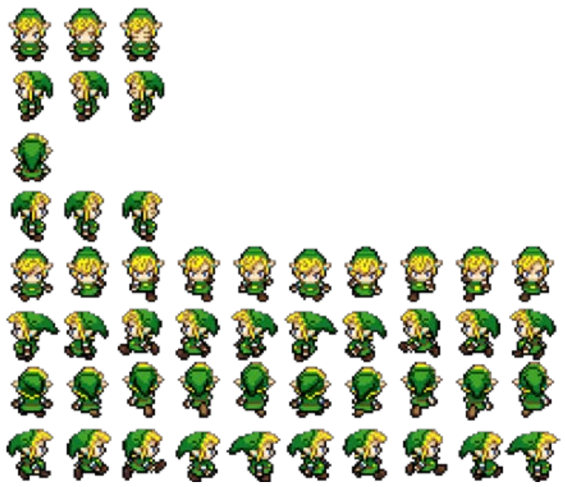


FIGURE 5.9. The first spritesheet we were able to import and use in Whiskey2D. (forum.unity3d.com)

2 / 2 / 2015

Improved Engine Features

Two major new features are completed. The existing collision algorithm is replaced by the Separating Axis Theorem (SAT). SAT supports Sprite rotation, as well as complex geometry. The collision system is greatly improved and simplified, giving designers access to collision Normals and Minimum Translation Vectors (MTV).

In addition, light and shadow support is added, as seen in Figure 5.10. The inclusion of dynamic lighting sees several improvements to Whiskey2D’s rendering pipeline.

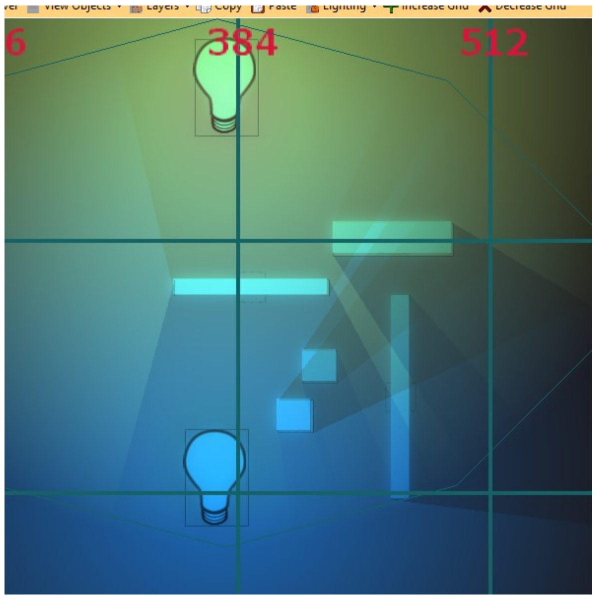


FIGURE 5.10. A screenshot showing the 2D lighting and shadowing system

2 / 24 / 2015

Usability Testing Round 3

We conduct our final round of usability testing with a near complete Whiskey2D. The users are able to complete the test and create a game. The users spend time tweaking and making their game fun to play. This is a huge success for Whiskey2D. (See Section 5.4 for details.)

3 / 11 / 2015

Whiskey2D

Development on Whiskey2D officially concludes. Minor edits to paperwork continue until 3/27, but Whiskey2D is complete.

3 / 26 / 2015

Showcase Game

Mechanical Plunge is completed, and demonstrates that Whiskey2D can be used to create complicated and entertaining 2D video games.

5.2 Development Environments and Tools

There were many tools that we used along the way when creating Whiskey2D. This includes many helpful pieces of software that helped us organize our documents and data, technology that helped facilitate communication, as well as tools for creation of media and source code.

Whiskey2D was written using Visual Studio[1], Microsoft's C# IDE. We also used Sublime Text[2], a text editor, for further source code creation, when writing some of our early game object scripts and for website code.

We utilized Git[3] for source control, which allowed us to keep our code up to date between each other, as well as allowing us to experiment with new features before we put them into our code-base. Along with Git, we utilized SourceTree[4], to help visualize and manage our Git repository.

In terms of media creation, we used GIMP[5] and Paint.NET[6]. These are both image editors, both similar in features to Adobe Photoshop. GIMP was, almost exclusively, used to

[1] More information for Visual Studio can be found at https://www.visualstudio.com/
[2] More information for Sublime Text can be found at http://www.sublimetext.com/2
[3] More information for Git can be found at http://git-scm.com/
[4] More information for Source Tree can be found at http://www.sourcetreeapp.com/
[5] More information for GIMP can be found at http://www.gimp.org/
[6] More information for Paint.NET can be found at http://www.getpaint.net/index.html

create various icons used within the editor. We used Paint.NET to create our many figures used in our weekly agendas, as well as many of the sprites we created for games created early on in development.

To manage all of our organizational materials and documents related to Whiskey2D, we created and shared a folder on Google Drive[1]. This service allowed us to save and store important files that we both needed to access. This is where our agendas, write-ups, user-testing files, and various task-management documents existed. We also created a website which hosted Whiskey2D information and information about our API.

The use of Facebook's[2] messaging service, helped facilitate communication. We were able to send text back and forth to settle on meeting times, as well as send screenshots and images to show progress or problems. We used this as our main source of communication throughout the development process. Another tool that helped with communication was the white-board located in our work area. This helped us express our thoughts to each other when describing concepts and ideas.

Finally, we used Doxygen[3] to help create documentation for Whiskey2D. With Doxygen, we were able to produce the files needed for a website. With this we were able to create a clean and user-friendly way to view Whiskey2D's structure and codebase for designers. In addition to the codebase documentation, we also created a few instructional videos for Whiskey2D using Camtasia Studio[4].

[1] More information for Google Drive can be found at https://www.google.com/drive/
[2] More information for Facebook can be found at https://www.facebook.com/
[3] More information for Doxygen can be found at http://www.stack.nl/~dimitri/doxygen/
[4] More information for Camtasia Studio can be found at http://www.techsmith.com/camtasia.html

5.3 Development Style

Lately, there have been many new software development styles emerging. Agile development is the most well-known among these modern styles, promoting practices like rapid prototyping, iterative development, scrum, and more. These new styles are meant to improve code quality and work productivity. We didn’t rigidly subscribe to any one particular development style, but instead pulled the concepts we liked from styles such as Agile or Rup.

We focused on following a few key development practices. The two biggest practices we followed were rapid prototyping and iterative development. The two theories go hand in hand with each other, promoting quick cyclic design. When we first started developing the Whiskey2D engine, we built the bare minimum to get a game up and running as quickly as possible. We were able to have a simple platformer game working through Whiskey2D’s engine within the first few weeks of development. Afterwards, we focused on strengthening our engine, and produced a new platformer in the following weeks. Developing the engine with an iterative approach allowed us to build the code out in a maintainable way. Once the basic engine was up and running, we shifted gears and started working on the editor backend code, as well as the editor’s UI. Developing the UI took place over several iterations, each one improving greatly over the previous. Finally, after we had a reliable editor, we went back to improving the engine’s feature set. We took a breadth first approach to building Whiskey2D that let us build up each part of the entire project piece by piece. If instead, we had focused solely on one aspect of development until it was complete, we never would have finished Whiskey2D to the point where it is now.

5.3.1 Meeting Flow

Whiskey2D was built by two people in about half a year. In order to finish the project, we met four to five times a week, on average. Our meetings were well structured and productive, allowing us to use our time efficiently.

Each week followed a rough development cycle. The first meeting of each cycle was spent mostly brainstorming. We would start by identifying a problem or feature that we needed to solve. Then we drew out diagrams and sketches for potential solutions. The brainstorming process was mostly done on a whiteboard that was in our work area. Since our work area was in one of our apartments, we were able to leave brainstorming sessions up on the whiteboard for the entire week. In many cases, we left large to-do lists up on the wall for many weeks.

After the brainstorming, the next part of our weekly design cycle was to write code. We used Peer Programming extensively, completing about 80% of our code together. Peer Programming is a recognized technique in software development where two people sit together, while one person does the typing, and the other monitors. The idea behind Peer Programming is that it forces each person to think hard about the code being written. The code has to make sense to both people as it is written down, so that the typist always has to stop and explain why they are doing what they are doing. As we were Peer Programming, we often realized that our original designs and ideas could be improved upon. Programming together helped keep a lot of hack solutions out of the code base, and encouraged strong implementations.

Finally, the last part of the weekly cycle was to meet with our project advisor, Professor Rich. The night prior to meetings, we created an agenda to guide our discussions for the following day. The agendas usually had a Section for a product demonstration, new feature technology and diagrams, weekly challenges, and finally what we intended to do in the following week. Agendas were sent to our advisor before each meeting so that he could be prepared to give us critical feedback on our progress. The weekly meetings helped us structure the challenges and goals for the next week. When the next week began, we started our process over again, starting with brainstorming on how to solve the problems discussed in the meeting.

5.4 Usability Testing

Ensuring that users can understand and use software is a critical aspect of software development. We held usability testing sessions to observe how people used Whiskey2D's editor. Primarily, we wanted to know how intuitive the editor was, and how easy it was to understand the basic game creation process. In order to answer our questions, we held three batches of usability testing.

5.4.1 Usability Testing Phase 1

The first session of usability testing happened early, around 7 weeks into development. In the first test, we asked our users to create a very simple game, where the player could control a smiley face around the screen, collecting stars. This can be seen in Figure 5.11. We created a website with simple instructions on how to create the game. In addition, we also created a simple API document that detailed all of the functions the user might need to use. All of the usability testing instructions can be found in Appendix B.

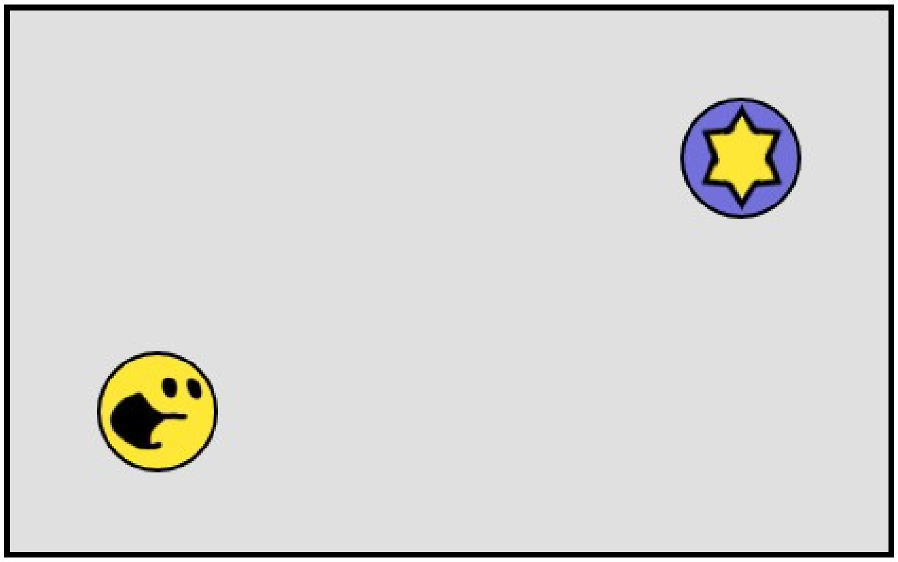


FIGURE 5.11. The first thing we asked users to create.

While we were excited to get feedback so early, our UI was not ready for users. We had only completed the first draft of a UI, and it was too buggy to be used in a usability test.

Although we knew that a UI refactor was necessary, we weren't aware how important it was. The main lesson we learned from the first phase of usability testing was that our UI was a deal breaker for users. It impeded the entire game creation process.

5.4.2 Usability Testing Phase 2

Several weeks later, we were able to completely redo the UI. After we had put enough functionality back into the editor, we began our second phase of usability testing. The goal of this usability test was to have users create a simplistic game of Breakout, as seen in Figure 5.12. This version of Breakout wouldn't have any blocks to hit, but it would support walls, collisions, simple physics, and player movement.

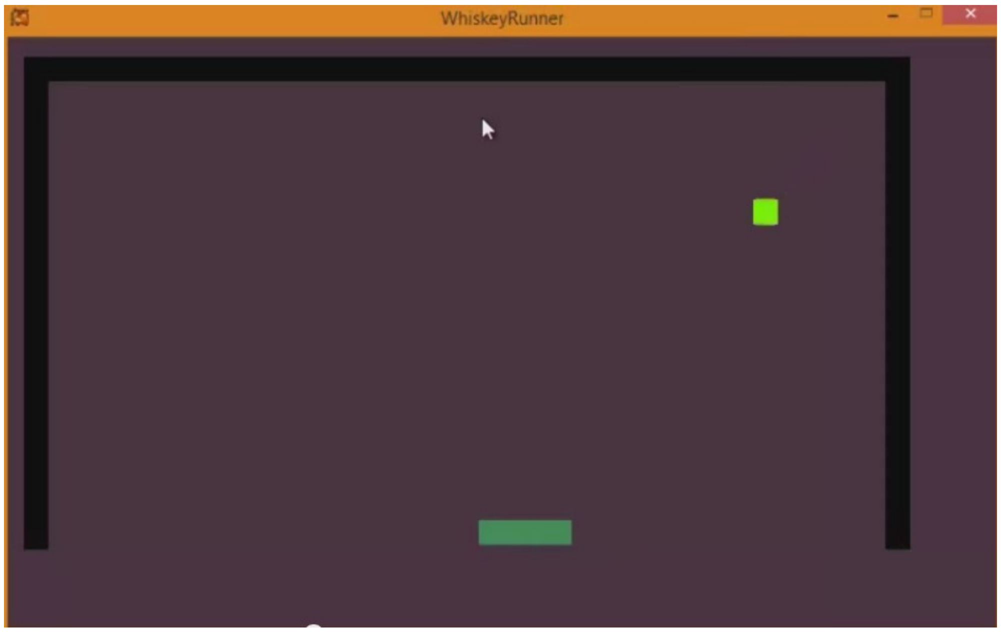


FIGURE 5.12. The first breakout game we asked users to create with Whiskey2D.

Instead of giving the users detailed instructions on how to create the game, we provided materials that explained how to use Whiskey2D in general, gave the users a prompt, and then asked them to complete it in a forty minute time period. These materials can be found in Appendix B. Additionally, instead of giving the users a wall of text to read, we created video explanations of Whiskey2D and of the game we wanted them to create. We began the usability test by asking users to watch a short YouTube video walkthrough[1] of how to use Whiskey2D. After that, we asked users to read a few paragraphs briefly explaining the core concepts discussed in Chapter 2. Finally, we showed them a video[2] of the game we wanted them to build. The videos were well received by the users, and they

[1] The instructional video we gave to our users can be viewed at https://www.youtube.com/watch?v=ubnPrt9-3Ng .
[2] The video prompt we gave the users can be viewed at https://www.youtube.com/watch?v=lb1TQsFKgaY

referred back to them often for guidance on how to achieve a particular task. Since we didn’t give the users a rigid set of steps to create the game, we were able to observe how they approached creating a game in Whiskey2D.

The improved UI was appreciated by our users, and they were able to focus on creating the Breakout level. However, the users expected common software features to exist, such as grid snap, copy and paste, along with a large collection of others. Each time a user tried to do something that wasn’t supported, they became irritated. However, these users were eventually able to create the Breakout level.

The users were confused when it came time for them to write the physics portion of the code. The user group we had did not match our designer profile discussed in Chapter 1. Instead of being amateur or hobbyist game developers, our users were complete beginners and had not written in C# at all. It is understandable that they struggled trying to write game logic code, but was disappointing because it prevented them from finishing the game. The main lessons we learned from the second usability test was that we needed to build out our editor more and make it more user friendly. We also learned that API documentation was going to be extremely important for designers.

5.4.3 Usability Testing Phase 3

For our last phase of usability testing, we were able to find people who had been making games with Unity, and were comfortable with game creation. We did our last usability testing at the end of our development time, within the last few weeks. The goal of our last usability test was to see if the changes we made from the previous test were effective. Like in the previous phase, we gave our users an instructional video to watch[1]. To our delight, the final usability test was a huge success. We asked the users to create an updated Breakout[2] game, seen in Figure 5.13. This time, the users were able to build the game. In fact, the users' Breakout game were better than our original demonstration game in the test instructions! The users were very impressed with Whiskey2D.

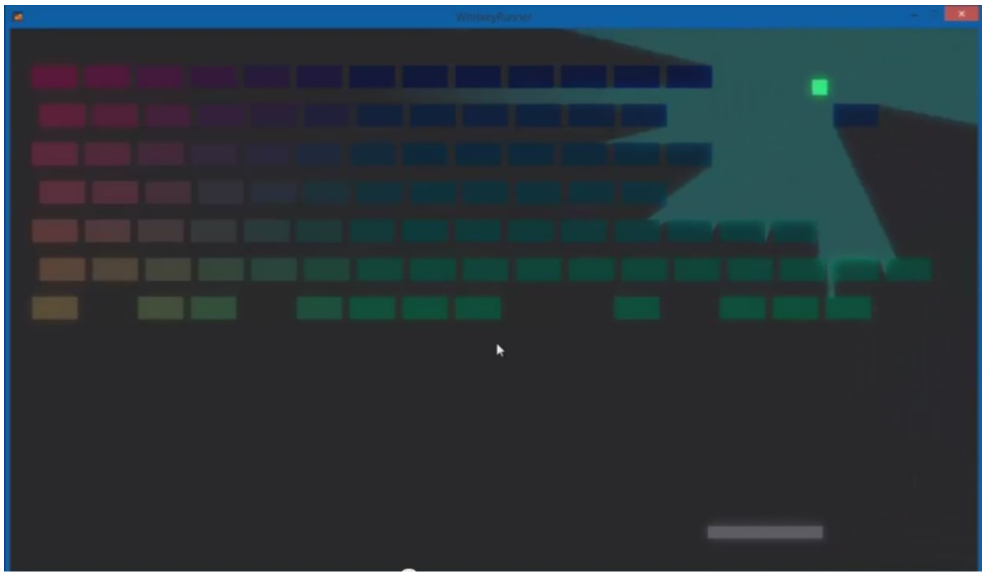


FIGURE 5.13. The second game of breakout made for usability testing

Despite the success of the test, users still requested additional features, such as group selection, undo/redo, and better code editing. As we were at the end of our development time, we weren't able to implement any of the features.

[1] The second instructional video we gave to our users can be viewed at https://www.youtube.com/watch?v=sVJHYpwkpFI

[2] The second video prompt we gave to our users can be viewed at https://www.youtube.com/watch?v=OwDXRLvwrjA

5.4.4 Usability Testing Results

We learned many valuable lessons through our usability tests. In our first phase, we saw that our UI was too weak. We had intentionally asked users to create a very simple game, but no matter how simple the task, our UI would have stood in the way. Our UI could not convey any of the core concepts to the users. This led us to a realization that our whole UI needed to be replaced.

In the second phase, we saw that our improved UI was able to convey core game creation concepts to the users. Users were finally able the use the UI. The new UI allowed users to learn more about GameObjects and Scripts, instead of how to simply use the editor. All of this led to users being able to identify specific feature requests such as grid snap and screen sizing. Our users did not fit the target audience profile, and this made the test more challenging. We were not aware that users out of the profile would have such a hard time.

In the final phase, we saw that Whiskey2D was usable! However, it was the first time we deployed Whiskey2D to non-development machines. Previously, we had only run Whiskey2D on our machines, and in this test, we ran it on WPI machines. The new machines caused new errors to appear in Whiskey2D that we had not seen before. However, the errors were not severe enough to deter the users from completing the test. Despite the errors, the users really enjoyed the Whiskey2D experience!

Each phase of the usability testing was useful. The tests gave us insights into Whiskey2D that we did not expect and might have overlooked. The results of each phase gave us direction for future development.

5.5 Showcase Game Design Process

Our showcase game, Mechanical Plunge, had a process of its own. We started by brainstorming ideas which we wanted to include in our game. This was important as we wanted to showcase the possibilities of our engine, and make sure that everything we put in was chosen to show a different aspect of the functionality included within Whiskey2D.

We met with an artist, Pat Gallagher, to brainstorm a simple, yet fun, game design. We sketched out many ideas on a white board, eventually settling on a platformer-shooter game. After our meeting with Pat, we developed some simple level designs on paper, and then worked to bring them to life inside Whiskey2D. Our first level designs can be seen in Figure 5.14, below.

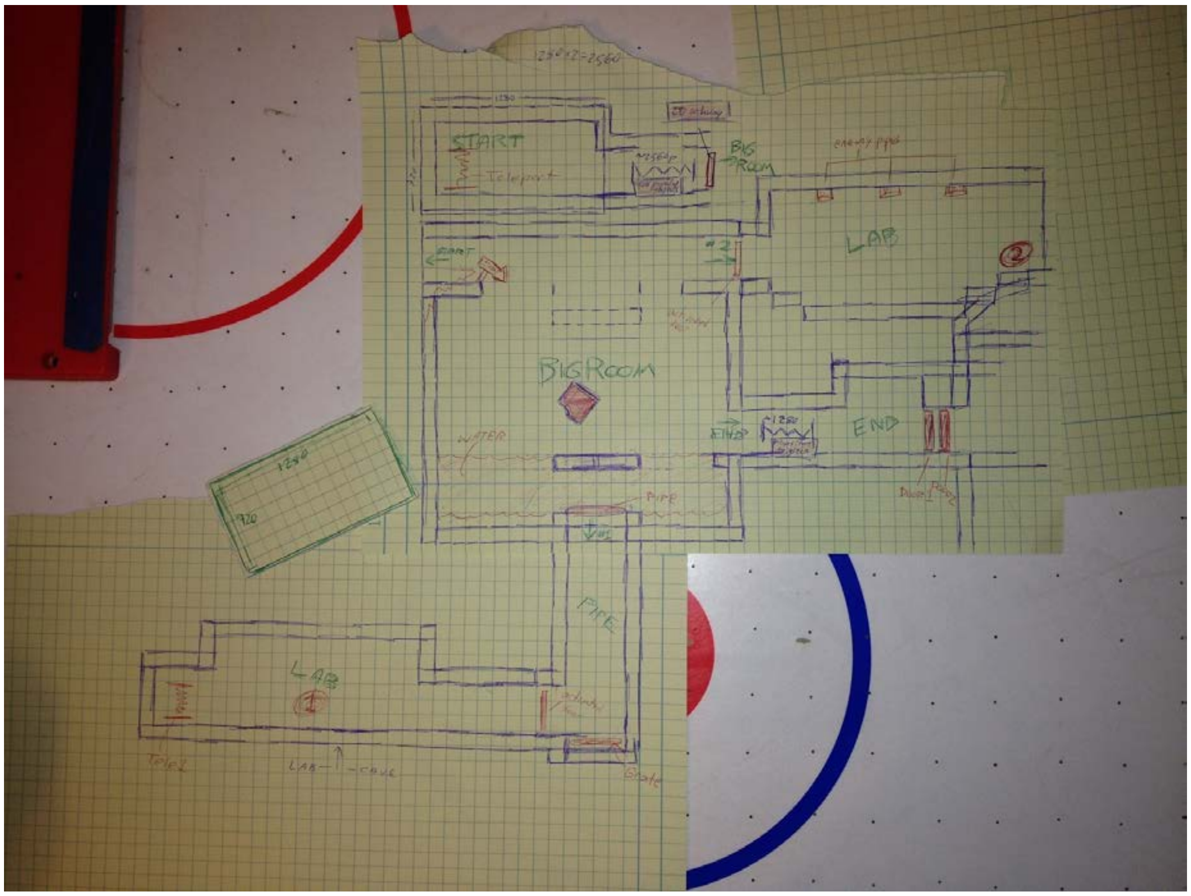


FIGURE 5.14. A picture of our original level design, done on grid paper.

Many hours were spent fine tuning game mechanics, and making sure that the game’s shooting and jump mechanisms felt right.

Pat Gallagher was able to assist us in art development. He created a sprite sheet for the main character of Mechanical Plunge. Another friend of ours, Bryce Rabideau was kind enough to compose an original game score. We communicated with our artists on a regular basis, and kept them up to date on the progress of the game.

Chapter 6 - Conclusion

Whiskey2D is in a stable working state[1]. Building Whiskey2D gave us terrific experience in creating software. We learned how to set technical goals, and achieve them. We learned about software engineering processes and how they're beneficial. We practiced our programming skills and improved our problem solving skills. However, all of this learning didn't come for free. We will shed light onto the greatest challenges we faced, both technical, and in our usability testing. We are also aware of at least two major limitations in Whiskey2D, both of which will be discussed in this Chapter. Finally, we will explain where we think Whiskey2D could be headed and what kind of work could be done in the future.

6.1 Challenges

As mentioned, developing Whiskey2D was no easy task. Many challenges were faced along the way both on the technical and on the process side of development. In this Section, we will discuss the largest technical challenges we faced, as well the challenges we faced during usability testing. A majority of the challenges we faced were overcome within a short amount of time and are not worth mentioning here. However, there were a few roadblocks that required a great amount of time and effort to circumvent.

[1] Whiskey2D can be downloaded from http://cdhanna.com/whiskey.

6.1.1 Technical

Dynamic Type Management

Perhaps the most troublesome dilemma we encountered during Whiskey2D's development was dynamic type management. A fundamental feature of Whiskey2D's editor was to support visual editing of GameObjects. As explained in Chapter 3, when a TypeDescriptor is dragged into the Level view, an InstanceDescriptor is created. The InstanceDescriptor describes an instance of the dragged TypeDescriptor. Using the InstanceDescriptor allowed us to modify properties of the Type without having to compile anything. All of the properties were represented by our data structures, instead of compiled class definitions. When the designer changed information about the Type, the system was able to pass those changes to all instances of the Type. The ability to change the Type definition and have all instances of the Type be updated is what we call dynamic type management.

The Descriptor system was not our first attempt at dynamic type management. Originally, when a Type was created or edited, it was immediately compiled to a C# class definition. The most recent Type compilation was then being used to instantiate a real instance of the Type. This technique gave us an incredible amount of power, because it meant that what the designer saw was exactly what the designer was going to get in the compiled game. If the designer went to the Type definition, and overrode the rendering function, then the instances of that Type would actually show the new rendering code running. However, using this approach spawned an incredible amount of complexity into our project.

One large problem with creating actual Instances of a Type was what to do when the Type definition changed. When a Type was edited, all of the Instances would have to be updated to reflect the change in the Type. For example, if the designer added a property to the Type, then all of the Instances of the Type would suddenly need to have the new property. Conceptually, all that needs to be done is to recreate every Instance using the newly compiled Type, sync the values of the new Instances to the old Instances, and then remove all of the old Instances. This process worked, but was ultimately ruined by the need for recompiling (see Figure 6.1)

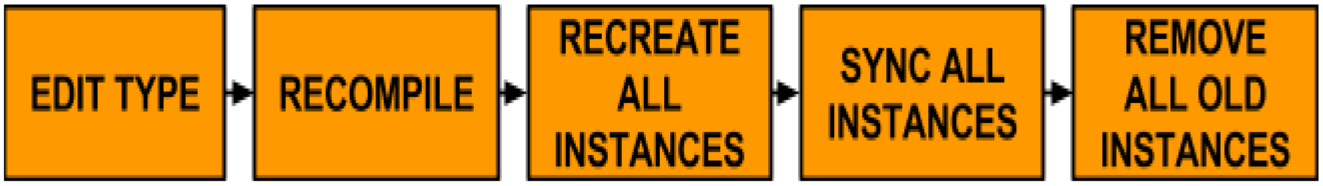


FIGURE 6.1. A process diagram showing the stages of syncing all instances to their modified type.

Recompiling the Type definition caused two major problems. Firstly, it was too slow. Changing the name of a property, its default value, or adding or removing a property took around two seconds to compile the change. Two seconds for every edit is far too long, even if it was put on a separate thread in the background.

The second problem with the compilation was how to load in the newly compiled Type information. The new Type information would be compiled to a DLL file. The DLL file would then be loaded into the editor's program memory so that it be used to create instances of the new Type. This worked, but had one serious flaw. Once a C# program loads Type information, the Type information cannot be unloaded. To be more specific, every C# program consists of at least one AppDomain. An AppDomain is a collection of Assemblies. An Assembly is collection of Type information. When a DLL is loaded, it comes in the form of an Assembly. Figure 6.2 shows this hierarchy of type management in C# below.

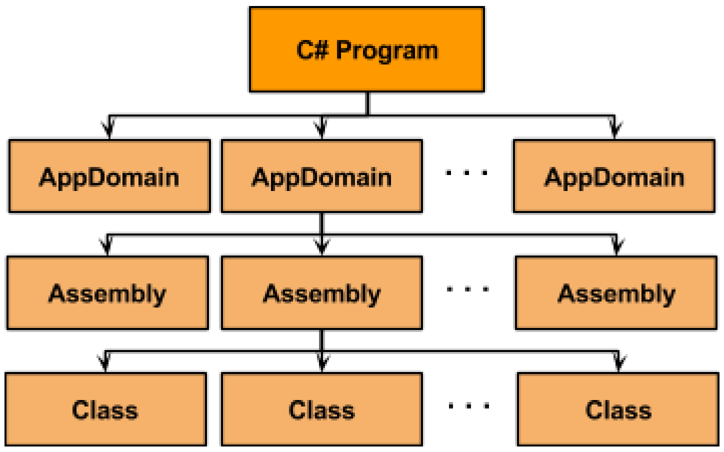


FIGURE 6.2. A breakdown chart showing the hierarchy of type management in a C# program.

Once the Assembly is loaded into an AppDomain, it cannot be unloaded from the AppDomain. This also means that the DLL file that actually contains the Type information cannot be deleted while the program is running, because the program has a lock on the DLL. If the DLL cannot be deleted, then when the designer makes an edit to a Type, and a new DLL needs to be compiled, it cannot overwrite the previous DLL. We generated new DLLs with version numbers appended them, and loaded each one into our program memory, without ever being able to unload the previous versions.

This caused a serious memory leak in our program. To correct for the leak, we created a system where a new AppDomain was created for each DLL. Every time we loaded a new DLL, we put it in a new AppDomain, and unloaded the previous AppDomain. This system worked, but it was incredibly slow. In addition, the infrastructure needed to pass data between AppDomains added too much complexity to our program.

In the end, we decided to reject the idea of using compiled Type information for our Instances. We spent weeks trying to make it work, and were reasonably close to a solution. However, to make the system run quickly enough to be usable was going to take too much time. Instead, we made the decision to delete week's worth of work in low level C# architecture design, and implement our Descriptor system instead. Our Descriptor system is almost as powerful, and works wonderfully.

User Interface, and Property Grid

Our other major technical challenge was working with Winforms, and specifically the built in PropertyGrid control. We started building our User Interface (UI) about a third of the way into our project. The UI was poorly designed and implemented. We combined too much of Whiskey2D's operating logic into the UI controls. The code was hard to manage, and not clean. Eventually, we realized that what we had for a UI was not good enough, and we began the UI refactor discussed in Chapter 5. Our second attempt at a UI was far better than our first.

However, in both versions of the UI, we struggled with WinForm's PropertyGrid control. The PropertyGrid has a rich set of features, and be customized in many different ways. We were unversed in the PropertyGrid, and it took us a long time to grasp how to leverage the control's full power. The PropertyGrid is the main control behind the Property View, and is central to Whiskey2D's editing experience. We were able to make the control behave like we wanted it to, but doing so took a major amount of our development time.

6.1.2 Usability Testing

Besides the large amount of technical challenges we overcame, we also encountered challenges associated with our usability testing. Our usability testing was done informally with friends and other students at Worcester Polytechnic Institute (WPI). As described in Chapter 5, we did three main passes of usability testing, spread through the development period of Whiskey2D.

When we first started showing Whiskey2D to our users, we showed them the original UI. As mentioned earlier, the original UI was poorly designed from the start. The only feedback we got from the testing were concerns about the quality of the UI. We tested too early, and didn't learn much from our test. In addition, our test instructions consisted of only text and pictures. Later, we found that videos worked much better.

In our second round of testing, our users didn't fit our designer profile, discussed in Chapter 1. The users didn't understand basic game programming logic, and spent too much time thinking about how to solve basic collision problems, instead of being able to focus on the substance of the Whiskey2D Editor. We weren't able to secure users that fit our designer profile until the end of development. Throughout the entire usability testing process, we struggled with our level of involvement with the users. In order to see how intuitive and usable Whiskey2D was, we tried to refrain from giving any hints to the users. However, in many cases, especially in the first phases of usability testing, we often helped users through the test scenarios.

6.2 Design Choice Limitations

In order to bring Whiskey2D to a state where it is stable and usable, we had to make some design decisions that brought limitations to the program. None of these limitations made the program unusable, though they are items that designers should be aware of.

Multiple Whiskey2D Instances

Firstly, only one instance of Whiskey2D may be running at a time. To be clear, when you are using your computer, you can open two copies of NotePad at once, and they are treated as separate instances. Whiskey2D only supports one running instance at a time. Technically, multiple instances could be run at once, but it would cause unpredictable behavior. Every Whiskey2D Project has its own file directory, but it also shares files from the main Whiskey2D editor directory. For example, when a designer loads Media into their project, the media must be copied into the Whiskey2D editor directory. If two instances of Whiskey2D were open with different Projects, media handling would become erratic, and files could potentially be overwritten. Running multiple instances of Whiskey2D should be avoided.

File Manipulation from Windows

The second major limitation Whiskey2D imposes is how files are to be manipulated. Our rule is that resource files for a Whiskey2D Project may only be manipulated from Whiskey2D's editor, and not from the Windows File Explorer. A designer can edit a source file with a text editor if they desire, but they cannot add files, remove files, or change file names. Whiskey2D keeps file information stored in a serialized state file. If the file system were to become out of sync with the state file, unpredictable behavior can occur, potentially resulting in a corrupted Project.

6.3 Future Development

There is still a lot of work to be done in Whiskey2D. We spent the majority of our development time building the core components of the engine and the editor. We weren't able to complete as many of the features as we wanted to. There are several algorithms that could be added, or be improved on, both in the core engine, and in the editor.

6.3.1 Engine Development

In order to complete the Whiskey2D engine, we took prioritized functionality over efficiency. Many of the algorithms use naive approaches, and could be re-implemented with better data structures and logic. For example, the ObjectManager uses a few lists to manage all GameObjects. The ObjectManager could have a dictionary of lists to make GameObject access faster. The collision system could use a spatial partitioning system such as quadtree, instead of using the worst case *n*-squared time algorithm it uses now.

Despite rapid development, there are still many features we'd like to see in Whiskey2D. There are several features common to most editors that are missing from Whiskey2D. The features can be added in one at time. For example, a particle system would be excellent to have built into the engine. Another incredibly useful feature would be built in collision response and physics. These features can be implemented by the designer, but it would be nice to have them supported by the engine.

6.3.2 Editor Development

In addition to adding more engine features, the Whiskey2D editor has much of room for improvement. There are many features that could be added, such as multiple object selection, object grouping, list editing, and more file manipulation such as renaming and movement. The code editor would be greatly improved by code completion. Finally, a better export feature that supported multi-platform game exporting would be fantastic.

We have mentioned a few features that we think would greatly improve Whiskey2D. There are clearly countless other features that would help build Whiskey2D into a professional tool. However, the core components of Whiskey2D have been implemented well, and are reliable. It is a tool that can produce non-trivial 2D games. Whiskey2D has a strong and robust back end that leaves the door open for future development.

Works Cited

[Statista 2015] *"Annual Revenue of the U.S. Video Game Industry by Segment 2014 | Statistic." Statista. N.p., n.d. Web. 21 Mar. 2015. <http://www.statista.com/statistics/249996/annual-revenue-of-the-us-video-game-industry-by-segment/>.*

[static2.scierra.net] *Construct2 Screenshot*. Digital image. *Static2.scirra.net*. Scirra, n.d. Web. 21 Mar. 2015. <https://static2.scirra.net/images/fresh/c2/gallery/fullsize/jpg/start-page-01.jpg>.

[i.solidfiles.net] *GameMaker Screenshot*. Digital image. *I.solidfiles.net*. YoYo Games, n.d. Web. 21 Mar. 2015. <http://i.solidfiles.net/cbc8a374e9.png>.

[forum.unity3d.com] *Link Spritesheet*. Digital image. *Forum.Unity3D*. Unity3D, n.d. Web. 21 Mar. 2015. <http://forum.unity3d.com/threads/2d-animation-issues.220428/>.

[video.ch9.ms] *Unity Screenshot*. Digital image. *Video.ch9.ms*. Unity, n.d. Web. 21 Mar. 2015. <http://video.ch9.ms/ch9/05b7/838999fc-16d4-4f59-9bda-9586a1bd05b7/Unity2013EndtoEnd_Custom.jpg>.

Appendix A: Whiskey2D Designers' Guide

All of the links below can be navigated to from Whiskey2D's home page, at http://cdhanna.com/whiskey/ .

Getting Whiskey2D Setup

To set up your Whiskey2D development environment, you'll need to download the latest release from *http://cdhanna.com/whiskey/*. If you'd like to develop Whiskey2D, or need to debug it, you'll need the complete source code. This can be cloned from Github, at https://github.com/cdhanna/whiskey2d. To make sure the solution builds correctly, follow the steps on the Github readme page.

Designer API Documentation

There are several built in features for you to use when writing Scripts in Whiskey2D. These features are documented at http://cdhanna.com/whiskey/docs/api/doxyapi/. The main features include the GameManager, the ObjectManager, and the InputManager. Other features include Rand, LogManager, and Sounds. For example usages of all features, download the sample showcase game, Mechanical Plunge, at https://github.com/cdhanna/whiskey2d/tree/master/samples/Showcase2

Shared Objects Listing

There are many built in GameObjects, provided with every Whiskey2D project. These are documented at http://cdhanna.com/whiskey/docs/sharedobjects/

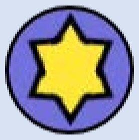
Appendix B: Usability Testing Instructions

Whiskey2D User Test 1

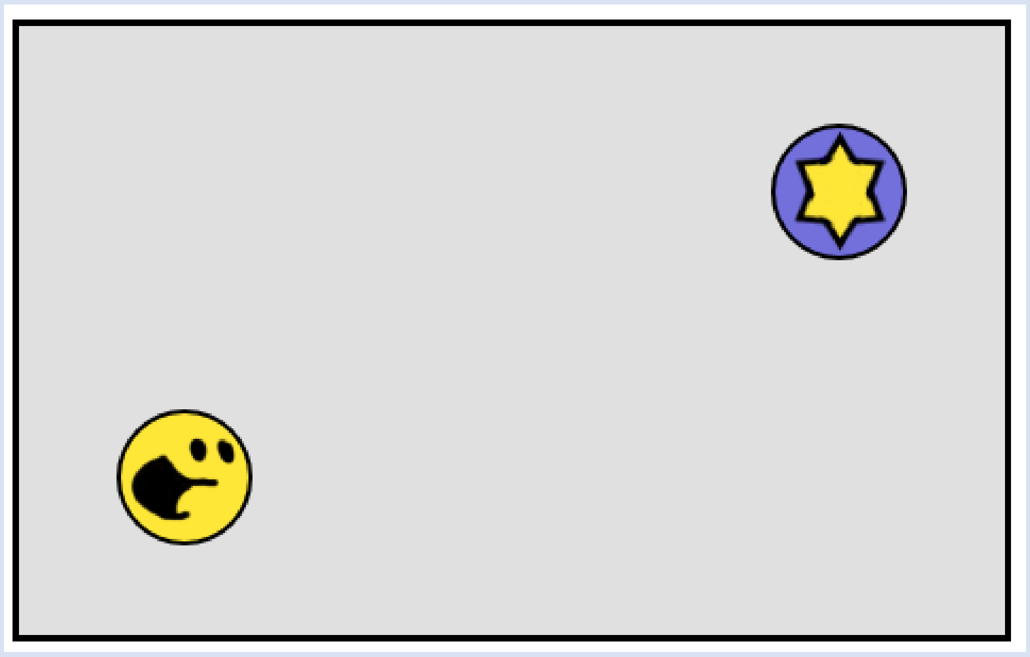
Please follow the steps in this simple tutorial to the best of your ability.
Refer to the included manual as needed.

1. Create a new GameObject type called Smile. Set Smile's sprite to use the smile.png image, located in the /media directory. Instantiate a Smile in the game space.

2. Create a new GameObject type called Star. Set Star's sprite to use the star.png image, located in the /media directory. Instantiate a Star in the game space.



Your gamespace should look like this…



3. Create a new Script called Move. Move is made to work with Smile objects.
   The Move script will allow the Smile object to move via the arrow keys. The code snippet below will move a GameObject to the right, when the right arrow key is pressed.

```
 if (GameManager.Input.isKeyDown(Keys.Right))
 {
     Gob.X += 1;
 }
```

   Using the above code, make the Move script allow for motion in all four directions. Add the Move script to the Smile instance.

4. Create a new Script called Collision. The onUpdate() code for Collision is below. Add Collision to the Star instance.

```
    public override void onUpdate()
    {
           Smile smileObj =
   GameManager.Objects.getAllObjectsOfType<Smile>()[0];
            //this will be cleaner in future versions
           if ( (smileObj.Position - Gob.Position).Length()
                < Gob.Sprite.ImageSize.X * .5f ){

            //code for step 4. Feel free to change this later, if you
want to
            Gob.X = Rand.getInstance().Next(100, 500);
            Gob.Y = Rand.getInstance().Next(100, 500);

            //todo: add Smile Code here (from step 6)

        }
    }
```

5. Hit the playTest button in the bottom-left of the screen, and go run your game. The exe will be generated in the /build directory.

Whiskey2D User Test 2

Thank you for helping us test Whiskey2D. The project has seen a lot of changes over the past few weeks, but we are still looking for critical design feedback. Our goal is to get your feedback on the Whiskey2D editor interface, and the usability of the Whiskey2D script API.

- First, take your seats and buckle up.
- Second, watch this video that details the basics of the Whiskey2D editor
- Now that you've seen the editor, its time to look at some of the core API components. Having a general idea of the API will help you solve the WHISKEY CHALLENGE that you'll be asked to complete in the end of this test.
  - ## GameObjects and Scripts

    The first thing to know about the Whiskey2D API, is what a GameObject is. As seen in the video, a GameObject is something you can place into a level. Everytime you drag in a GameObject, the system is creating a new instance of that GameObject.

    Scripts, on the other hand, are what control GameObjects. GameObjects hold a set of these. In the video, the GameObject just had some data inside of it, such as an (X, Y) position. The Script is what actually *moved* the GameObject, by modifying the (X, Y) location.

    So, GameObjects hold data, and Scripts update data. A Script accesses the GameObject's data by calling code that looks like this… (this is code in the Script itself, and Gob is a field in Script)

    ```
    this.Gob.X += 1; //increment the GameObject's property, X
    ```

You can define your own properties in your GameObjects, but every GameObject has an X, Y, Sprite, and a Bounding box (we'll talk about this later)

Every Script has two methods in it, an *onStart()* and an *onUpdate()*. The *onStart()*method is called at the very beginning of the game, and the *onUpdate()* method is called once every tick. Most of the logic in Scripts get implemented in the*onUpdate()* method.

- ### GameManager services

The GameManager provides a set of services that can be used from Scripts to aid in general game coding. For now, we are only going to discuss the Input service. The Input service provides a set of functions to check various states of keyboard and mouse input. In the video, you saw how to check if the 'Right' key was down. To check other keys, use a similar approach…

```
if (GameManager.Input.isKeyDown(Keys.Left)){
    //the left key is being pressed!
}
```

- ### Collisions

To check if two GameObjects are colliding, there is a handy function called*currentCollisions* . Calling this function from a script will get all of the other GameObjects that something is colliding with.

```
List< Thing > collidingThings = Gob.currentCollisions< Thing >();
```

The above code will get all of the Things that the Gob is colliding with. (remember that Thing was the type from the video) This will be very helpful for the upcoming WHISKEY CHALLENGE!!!!

- ## Whiskey Challenge

Thanks for sticking with us through all that text.

Now, we want you to build a game that looks like this…

For a (almost) full documentation set, go here

Here are some hints to help you get started…

You'll need a Level

When we built this, we used 3 GameObject types, and 2 Scripts

The 3 GameObjects were

```
*    Player
*    Wall
*    Ball
```

The 2 Scripts were

```
*    PlayerScript< Player >
*    BallScript< Ball >
```

Start simple, build the player and wall collisions first. Then move onto the ball.

When things start colliding with Walls, you may find it very helpful to add a property to the Wall GameObject, called 'Name', and use that to figure out WHICH wall is being collided with.

Feel free to ask us any questions!

Thanks again!!!

Whiskey2D User Test 3

Thank you for helping us test Whiskey2D. The project has seen a lot of changes over the past term, but we are still looking for critical design feedback. Our goal is to get your feedback on the Whiskey2D editor interface, and the usability of the Whiskey2D script API.

- First, take your seats and buckle up.
- Second, watch this video that details the basics of the Whiskey2D editor
- Now that you've seen the editor, its time to look at some of the core API components. Having a general idea of the API will help you solve the WHISKEY CHALLENGE that you'll be asked to complete in the end of this test.
    - ## GameObjects and Scripts

      The first thing to know about the Whiskey2D API, is what a GameObject is. As seen in the video, a GameObject is something you can place into a level. Everytime you drag in a GameObject, the system is creating a new instance of that GameObject.

      Scripts, on the other hand, are what control GameObjects. GameObjects hold a set of these. In the video, the GameObject just had some data inside of it, such as an (X, Y) position. The Script is what actually *moved* the GameObject, by modifying the (X, Y) location.

      So, GameObjects hold data, and Scripts update data. A Script accesses the GameObject's data by calling code that looks like this… (this is code in the Script itself, and Gob is a field in Script)

      ```
      this.Gob.X += 1; //increment the GameObject's property, X
      ```

      You can define your own properties in your GameObjects, but every GameObject has an X, Y, Sprite, and a Bounding box.

Every Script has three methods in it, an *onStart()* , an *onUpdate()*, and an *onClose()*. The *onStart()* method is called at the very beginning of the game, and the*onUpdate()* method is called once every tick. Most of the logic in Scripts get implemented in the *onUpdate()* method.

- ## GameManager services

The GameManager provides a set of services that can be used from Scripts to aid in general game coding. For now, we are only going to discuss the Input service. The Input service provides a set of functions to check various states of keyboard and mouse input. In the video, you saw how to check if the 'Right' key was down. To check other keys, use a similar approach…

```
if (GameManager.Input.isKeyDown(Keys.Left)){
    //the left key is being pressed!
}

If (GameManager.Input.isKeyDown(Keys.Right)){
    //the right key is being pressed!
}
```

- ## Collisions

To check if two GameObjects are colliding, there is a handy function called*currentCollisions* . Calling this function from a script will get all of the other GameObjects that something is colliding with.

```
Collisions< Wall > collidingWalls = Gob.currentCollisions< Wall >();
```

The above code will get all of the Things that the Gob is colliding with. (remember that Wall was the type from the video) This will be very helpful for the upcoming WHISKEY CHALLENGE!!!!
Like in the video, the result of *Gob.currentCollisions< Wall >()* is a collection of objects. Usually, a good way to handle collisions is to iterate through all of them like the code segment below.

```
foreach (Collision<Wall> c in collidingWalls){
    //process collision, c
}
```

Notice that the word, *Collision* is used in the foreach statement, and not the collection type, *Collisions*.

- ## **Whiskey Challenge**

Thanks for sticking with us through all that text.
Now, we want you to build a game that looks like this…

For a (almost) full documentation set, go here
Here are some hints to help you get started… We do not expect you to create all of the bricks, so just having the ball moving around the level and colliding with walls and the player is good enough.

You'll need a Level

Start simple, build the player and wall collisions first. Then move onto the ball.

Remeber to use the Vector's *reflect* function. Check the API!

Feel free to ask us any questions!
Thanks again!!!

Appendix C: Development Videos

We created several videos throughout the development of Whiskey2D to demonstrate various features to each other, and to friends. All of our development videos were posted to YouTube, and can be found below.

- Usability Test Phase 2, Video Prompt
  https://www.youtube.com/watch?v=lb1TQsFKgaY
  The video prompt we gave our users in the second usability test.

- Usability Test Phase 2, Video Instructions
  https://www.youtube.com/watch?v=ubnPrt9-3Ng
  The instructional video we asked users to view in second test.

- Simple Image Support
  https://www.youtube.com/watch?v=1WLdY2aHSJY
  A video showing off image support for Sprites.

- Scaling and Shaders
  https://www.youtube.com/watch?v=swtz7DCJPT4
  A video showing off new ways to edit GameObjects visually, as well shader controls for a Level.

- Level Creation Time-lapse 1
  https://www.youtube.com/watch?v=Z0NVxQd5BuQ
  A time-lapse of a simple Whiskey2D level being created.

- Level Creation Time-lapse 2
  https://www.youtube.com/watch?v=uHfr5CUPL24
  A time-lapse of a simple Whiskey2D level being created, now with lighting!

- Usability Test Phase 3, Video Prompt
  https://www.youtube.com/watch?v=OwDXRLvwrjA
  The video prompt we gave our users in the third usability test.

- Usability Test Phase 3, Video Instructions
  https://www.youtube.com/watch?v=sVJHYpwkpFI
  The instructional video we asked our users to view in the third test.

- Camera Testing
  https://www.youtube.com/watch?v=eIbLrQLqGFM
  A video demonstrating the CameraMaster tracking a GameObject through CamZones.

- Mechanical Plunge Simple Platforming
  https://www.youtube.com/watch?v=Pf1GxxLAyw4
  A video showing the basic layout of our initial Level design.

- Mechanical Plunge Shooting Example
  https://www.youtube.com/watch?v=AMCTx-5ydtM&feature=youtu.be
  A video showing the shooting effect for the character.

- Mechanical Plunge Character Animation
  https://www.youtube.com/watch?v=sCdRPE_y0BU
  A video showing the animation system for the character.

- Mechanical Plunge Water Physics
  https://www.youtube.com/watch?v=haUMoFetAcs
  A video showing water inside our game.

- Mechanical Plunge Water Physics 2
  https://www.youtube.com/watch?v=dPIWke2MlxI
  A video showing some water refinements.

- Mechanical Plunge Game Events
  https://www.youtube.com/watch?v=4KgDZxSmLW8
  A video showing how a button can control various things in a Level.