**Technical Document of Game Project *Lost***

**Developed by:**

Clay Westing

Deblin Sarker

Emi Eve

Khadeidre Dean

Trevor Newell

Zhitao Zhao

**09/11/2017**

**Technical Document of Game Project Lost**

* **Introduction**

Lost is a 2D maze game developed by Monogame Engine. It uses a top-down visual angle. Players use a gamepad to control a frog to finish a maze in about 150 seconds. The player’s view is limited and it will reduce with the time.

This document introduces the concrete implementation of the game. It lists all the useful classes and interfaces. Although this is not a complex game, some tools can be used in next projects. The document tries to represent important systems as thoroughly as possible to make sure that the project can be modified easily. Common mistakes are also recorded as references.

* **Developers guide**

This part is for all the developers including producers, artists, engineers, and technical artists.

**2.1 Develop Environment**

Lost is developed by Monogame engine. MonoGame is an Open Source implementation of the Microsoft XNA 4 Framework. To further develop the game, developers should install Monogame engine from:

<http://www.monogame.net/downloads/>

We recommend developers to use Monogame for Visual Studio.

**2.2 Game Resources Requirement**

All the game resources should be imported by pipeline tool. The Pipeline Tool is used to organize and build content for use with MonoGame. Developers can find how to use Pipeline Tool from:

<http://www.monogame.net/documentation/?page=Using_The_Pipeline_Tool>

Different resource files should be placed in their respective paths. Images that include maps, blocks or GUIs should be placed in Graphics folder. Audio files should be placed in Sounds folder.



Assets should be in the following formats:

* Image files should be .png format
* Background music files should be .mp3 format
* Sound effect files should be .wav format

The project use csv files to save game data. They can be easily edited in Excel or Notepad.

<https://en.wikipedia.org/wiki/Comma-separated_values>

Level designers can write image’s position (left-up point) in the files so that the images will show up in the right place of the game map. These Csv files can be found in:

Gameproject\bin\Windows\x86\Debug\Content\Data

* **Components and Functions**

This part is for engineers and technical artists. It introduces main classes and interfaces of the project. Important functions will be explained in theory.

* **Main**

Game1.cs is the main class. Here is the main game loop. API can be found in:

<http://www.monogame.net/documentation/?page=T_Microsoft_Xna_Framework_Game>

We recommend that classes that need to be drawn on screen can also have these four functions so that the main loop can just call these classes’ corresponding functions. In this way, the code in the main class will be more concise.

For example:



Game input is in Update() function. Player uses the left stick of the controller to move the frog. Inputs are values on the X-axis and Y-axis. Use these two values to calculate the rotation of the frog(by trigonometric function) so that we can use the stick to change frog’s speed and direction. Note that the return of trigonometric function is between 0 and .

* **Player**

Class player and any other class that can be drew on the screen such as block, enemy and GUI can be a subclass of Entity. Regrettably we did not do that this time.

* **Movement**

Monogame engine doesn’t have physical engine. It only provides position of sprite. As our game is in top-down visual angle. We use a velocity vector to calculate frog’s speed in X-axis and Y-axis.

velocity.X = (float)Math.Sin(rotation) \* speed;

velocity.Y = (float)Math.Cos(rotation) \* speed;

We can get next position by adding position vector and velocity vector. Still attention that the return of trigonometric function is between 0 and .

Position.X += velocity.X;

Position.Y -= velocity.Y;

* **Collision Detection**

The project uses both rectangle detection and pixel detection to detect collision between player and blocks because most of blocks have irregular shape. First, detecting if player’s rectangle intersects with a block’s rectangle. We can use class Rectangle’s function to implement it.

<http://www.monogame.net/documentation/?page=M_Microsoft_Xna_Framework_Rectangle_Intersects_1>

If two rectangles intersect, we call DetectPixelCol(Block block). There will be a new rectangle called intersecting rectangle between two rectangles. Traverse all the pixels in it. If there is a pixel that both the player and the block’s alpha values are more than zero(or some other values), it means the player collides with the block.

* **Animation**



Animation within Monogame involves iterating through a sprite sheet (pictured above) that's defined as a Texture2D type.

The following line determines what sprite and spritesheet to use for our animation:

player.Initialize(Content.Load<Texture2D>("Graphics\\player"), Content.Load<Texture2D>("Graphics\\hopSheet"), playerPosition);

\*Note, our player class is initialized within Game1.cs.\*

The "Graphics\\player" textures is just the first frame of our spritesheet, isolated in its' own file. The "Graphics\\hopSheet" is the sprite sheet pictured above.

The following is updated each tick from within our Update(Gametime gameTime) function:

private void UpdateAnime(GameTime gameTime)

{

previousKBState = currentKBState;

currentKBState = Keyboard.GetState();

if (currentKBState.IsKeyDown(Keys.Up))

AnimateHop(gameTime);

if (currentKBState.IsKeyDown(Keys.Down))

AnimateHopBackwards(gameTime);

}

This allows us to advance forward (when the up arrow is pressed) or backwards (when the down arrow is pressed) through our spritesheet.

public void AnimateHop(GameTime gameTime)

{

// If we're just barely starting to move set our frame to the first image of our hop cycle

if (currentKBState != previousKBState)

{

currentFrame = 0;

}

timer += (float)gameTime.ElapsedGameTime.TotalMilliseconds;

if (timer > interval)

{

// Advance our frame forward in our spritesheet

currentFrame++;

// If we've reached the end of our spritesheet, reset to the start of the spritesheet.

if (currentFrame > end - 1)

{

currentFrame = 0;

}

timer = 0f;

}

}

public void AnimateHopBackwards(GameTime gameTime)

{

// If we're just barely starting to move set our frame to the last image of our hop cycle

if (currentKBState != previousKBState)

{

currentFrame = end;

}

timer += (float)gameTime.ElapsedGameTime.TotalMilliseconds;

if (timer > interval)

{

// Advance our frame backwards in our spritesheet

currentFrame--;

// If we've reached the beginning of our spritesheet, reset to the end of the spritesheet.

if (currentFrame < 0)

{

currentFrame = end - 1;

}

timer = 0f;

}

}

With the comments in the code, it's pretty self explanatory.

* **Camera**

In Monogame engine, there is not a concept of camera. We use a simple 2D matrix to implement it. The document of Matrix can be found here:

<http://www.monogame.net/documentation/?page=T_Microsoft_Xna_Framework_Matrix>

Monogame Engine uses spritebatch class to draw sprites. In the function Begin(), a parameter transformMatrix can transform the sprite geometry. We can set camera’s matrix here so that the sprite’s transform, rotation and scale will change.

<http://www.monogame.net/documentation/?page=M_Microsoft_Xna_Framework_Graphics_SpriteBatch_Begin>

We know that if the frog goes right, the background can be regarded as going left while the player is still. So just drawing other sprites in the opposite direction to the frog can make the camera looks like following the frog.

transform = Matrix.CreateTranslation(new Vector3(-center.X, -center.Y, 0)) \*

Matrix.CreateRotationZ(rotation) \*

Matrix.CreateScale(new Vector3(zoom, zoom, 0)) \*

Matrix.CreateTranslation(new Vector3(viewport.Width / 2, viewport.Height / 2, 0));

* **GUI**

GUI can also be regarded as subclass of Entity. The only difference is that GUI must follow camera. So just set GUI’s position with camera’s position. But don’t forget the camera has transform, rotation and scale. Make sure that GUI also follow these matrix translations.

* **setOffset()**

This function is to set the offset between a GUI sprite’s position and camera’s position.

public void setOffset(int X, int Y)

{

offset.X = X;

offset.Y = Y;

}

* **CSV Tool**

The project use csv files to save game data. In this way, level designer can modify game data easily without the help of engineers. To use the data, we must transfer it into Datatable object. Datatable is a convenient data table class in C#. Document about Datatable can be found here:

<https://msdn.microsoft.com/en-us/library/system.data.datatable(v=vs.110).aspx>

* **readCSV()**

In a csv file, data is split by commas. So just detect all the commas in a line and we can get all the data.

unitsInLine = lineString.Split(',');

First line is different, there are titles of rows. We need to use these titles to create columns in the data table.

if (isFirstLine == true)

{

columnCount = unitsInLine.Length;

for (int i = 0; i < columnCount; i++)

{

DataColumn dataColumn = new DataColumn(unitsInLine[i]);

dataTable.Columns.Add(dataColumn);

}

After that, we can use the same method to split rest lines and save them in datatable object’s rows.

DataRow dataRow = dataTable.NewRow();

for (int j = 0; j < columnCount; j++)

{

dataRow[j] = unitsInLine[j];

}

dataTable.Rows.Add(dataRow);

Then we can use datatable to control data such as setting position and so on.

* **Glow Effect**

The glow effect is achieved through the use of an "Effect.fx" file called "lighteffect.fx":

sampler s0;

texture lightMask;

sampler lightSampler = sampler\_state{Texture = <lightMask>;};

float4 PixelShaderFunction(float4 pos : SV\_POSITION, float4 color1 : COLOR0, float2 coords: TEXCOORD0) : COLOR0

{

float4 color = tex2D(s0, coords);

float4 lightColor = tex2D(lightSampler, coords);

return color \* lightColor;

}

technique Technique1

{

pass Pass1

{

PixelShader = compile ps\_4\_0 PixelShaderFunction();

}

}

\*\*\*\*\* Documentation for this function is outdated. If you want to redact one of the parameters in the following function header:

float4 PixelShaderFunction(float4 pos : SV\_POSITION, float4 color1 : COLOR0, float2 coords: TEXCOORD0) : COLOR0

It has to be the last one(s). If you remove one of the first ones like so (and like how the documentation has it listed):

float4 PixelShaderFunction(float2 coords: TEXCOORD0) : COLOR0

It will NOT work. Keep the header like it is in the "lighteffect.fx" file and you won't have any issues.

You'll need the following variables to use the lighteffect.fx file properly:

Texture2D lightMask;

RenderTarget2D lightsTarget;

RenderTarget2D mainTarget;

Effect lightingEffect;

Initialized like so:

lightMask = Content.Load<Texture2D>("Graphics\\sampleLightMask");

lightingEffect = Content.Load<Effect>("lighteffect");

var pp = GraphicsDevice.PresentationParameters;

lightsTarget = new RenderTarget2D(

GraphicsDevice, pp.BackBufferWidth, pp.BackBufferHeight);

mainTarget = new RenderTarget2D(

GraphicsDevice, pp.BackBufferWidth, pp.BackBufferHeight);

Now all of the other logic happens within the Draw(GameTime gameTime) method.

**We render all of our light masks to the same target (think of it as a panel)**

GraphicsDevice.SetRenderTarget(lightsTarget);

GraphicsDevice.Clear(Color.Black);

spriteBatch.Begin(SpriteSortMode.Immediate, BlendState.Additive, null, null, null, null);

**These are our lights. One that follows the frog and shrinks. The other highlights the ball.**

spriteBatch.Draw(lightMask, new Vector2(player.Position.X - ((lightMask.Bounds.Width / 2) \* scale), player.Position.Y - ((lightMask.Bounds.Height / 2) \* scale)), null, Color.White, 0, Vector2.Zero, scale, SpriteEffects.None, 0f);

spriteBatch.Draw(ballGlow, new Vector2(ball.position.X - ballGlow.Bounds.Width/5, ball.position.Y - ballGlow.Bounds.Height/5), null, Color.White, 0, Vector2.Zero, 0.4f, SpriteEffects.None, 0f);

spriteBatch.End();

**Now that we've rendered our lightmasks onto our lightsTarget. We can now render everything else that will be affected by the lighting onto our "mainTarget".**

GraphicsDevice.SetRenderTarget(mainTarget);

GraphicsDevice.Clear(Color.CornflowerBlue);

spriteBatch.Begin(SpriteSortMode.Deferred, BlendState.AlphaBlend, null,null,null,null);

spriteBatch.Draw(mainBackground, rectBackground, Color.White);

player.Draw(spriteBatch);

ball.Draw(spriteBatch);

wall.Draw(spriteBatch);

spriteBatch.End();

**Now we have two panels setup. One with our lightmasks in the correct spots, and one with everything else in our game (except the UI.. not effected by light) in the correct spots.**

// Draw the main scene with our pixel shader being applied

GraphicsDevice.SetRenderTarget(null);

GraphicsDevice.Clear(Color.CornflowerBlue);

spriteBatch.Begin(SpriteSortMode.Immediate, BlendState.AlphaBlend);

lightingEffect.Parameters["lightMask"].SetValue(lightsTarget);

lightingEffect.CurrentTechnique.Passes[0].Apply();

spriteBatch.Draw(mainTarget, Vector2.Zero, Color.White);

spriteBatch.End();

**Now that we've rendered in our lighting and the main scene we can draw our UI**

// UI

spriteBatch.Begin(SpriteSortMode.Deferred, BlendState.AlphaBlend, null, null, null, null, camera.transform);

currentGUI.Draw(spriteBatch, camera.center);

if (gameStarted == true)

healthBar.GUIRectangle.Width = (int)player.health;

healthBar.Draw(spriteBatch, camera.center);

healthBarBorders.Draw(spriteBatch, camera.center);

spriteBatch.End();

base.Draw(gameTime);

}