Legend of Iris

Game Technical Report

Version 1.0
16th of January 2015

Bas Dado
Kevin Allain
Mick van Gelderen
Miguel Oliveira
Olivier Hokke

This report describes and justifies the technical choices that have been made for the final prototype of Legend of Iris.

Game Engine

We have decided to use the free to use game engine Unity (http://unity3d.com/). The reason we choose this engine is that it is a very flexible engine that functions as a great easy-to-use and easy-to-learn sandbox. While the core is lightweight, there is a so called asset store which contains use made packages of 3D models, textures, scripts and more. The unity editor itself is written using the Unity system and, because of that, can be easily extended and tailored to your needs. Another great thing about Unity is that the games you create with it can be built for almost all popular PC and mobile operating systems, and even game consoles.

Target Platform

Legend of Iris is designed to be played on a PC running Windows, Mac OS X or Linux. While it would be possible to adapt the controls of our game to mobile platforms, sound simulation requires quite some computational power which might not be present. While creating a mobile version is opens up our game to an extremely big audience, it is not something we focus on for the prototype.

Controls

The most basic setup for Legend of Iris is a computer and a keyboard, you don't even need a monitor. While this setup is sufficient to play the game, plugging in a game controller will provide a more natural way of commanding your character. To further improve, we decided to provide support for the Oculus Rift for its head-tracking capabilities which turned out to be a huge success.

Sound

In real life, every sound is affected by the direction and velocity at which the originating source moves, as well as its current distance from the listener. Furthermore, all objects reflect and absorb sound, which creates reverb and occlusion effects. This means that even the shape and material of the body, head and ears affect sounds before they reach your ear drums.

After some initial research, it turned out that having real-time realistic reverb and occlusion effects based on the actual scene is very hard. We found one library, called GSound (http://www.carlschissler.com/gsound/), that claimed to be able to do it. However, this library was based on C++ and showed some artifacts, such as clicking sounds when the listener moved, when used on slower hardware. Writing a wrapper to use GSound inside Unity took more time than we allowed ourselves to spend on it, so we decided to look for other options.

We also experimented with creating our own occlusion engine in Unity. It was based on the idea that it could be roughly approximated using the NavMesh system, where many different paths of different lengths are found from the source to the listener. Each path then would represent some propagated wave. For each path we calculated the length and whether or not it passed through a wall. The length would then define the delay and create a reverb effect. Sound on paths that went through walls would be low pass filtered, so it sounds occluded. This system quickly appeared to have a lot of deficiencies. When we wanted to move listeners and audio sources, cracking noises were sometimes introduced. Creating a real-time sound simulation engine is more of a separate project.

Another library that promises to deliver realistic scene-based occlusions is Phonon SoundFlow (http://phonon.impulsonic.com/products-phonon-soundflow/). However, this library is not yet available to the public, although it is expected to be released later in 2015.

For our gameplay, being able to localise the sound is more important than having (approximately) correct occlusions and reverbs, we have decided to focus on more realistic surround sound instead of realistic occlusions and reverbs.

One well known technique to get semi-realistic surround sound through stereo headphones is the head-related transfer function (HRTF). An HRTF is a filter that, when a applied to a 3D sound, transforms it such that it sounds similar to the way perceived if the sound was actually located somewhere in space. It includes transforms in the sound caused by the body, head, and ears of the listener. There are several libraries available that can apply the head-related transfer function to a sound.

The library that is mainly featured in our final product is AstoundSound (http://www.astoundgaming.com), which was kindly provided to us for educational purposes by the AstoundSound developers. AstoundSound claims to use a so-called brain-related transfer function,

that, based on years of research, transforms the sound to the way the brain perceives it. This should allow the player to locate any sound source using stereo headphones.
We also offer the option of using the (still experimental) library Phonon 3D (http://phonon.impulsonic.com/products-phonon-3d/). This library also offers a head-related transfer function, and is very scalable in performance, applying the needed transforms only on the sounds that are close to the listener.

The last choice we offer the users is to use Unity's own sound library. The Unity sound library has the advantage to be native to the unity engine, and offers stereo sound and possibilities to modify how the sound loses it's volume through distance. The big issue with Unity native sound library is that sounds behind you sound exactly the same as when it would have been played in front of you since the sound is only being panned. Players can still determine where it is eventually by moving and/or rotating. Getting the audio as realistic as possible was of great importance however, so the default sound engine is not recommended.

User interaction

During our beta, we used to use classic First Person Shooter controls, which are moving the body with WASD and moving the head with the mouse. Conversing with blind people informed us that they almost never use the mouse as it is very difficult for them to locate the cursor on the screen during a regular use of the computer.

Another important point we took into consideration during the beta was that the player tried to first locate where the sound came from, rotate to face it, and then move forward in order to reach it. Since the player only tries to move forward, it seemed more logical to modify the controls to make rotations easier even though that meant dropping the easiness to make side steps.
Because of this, we decided to offer several possibilities with the controls:

- Single Handed: Single handed controls allows the user to move and turn using a keyboard WASD or arrows, or a Xbox 360 game controller. Forward and backwards are used for moving, while left and right arrows (or S and D keys) can be used for turning.
- FPS: As said before, the mouse is used to represent the head and the keyboard moves the character forwards, backwards and sideways.
- Relative turns: This controls setting requires a controller. The left analog stick acts the same way as it would in the FPS controls. The right analog stick can be used to turn the character, but in a very precise way. The character turns relative to the current forward direction. The player should aim the right stick to the direction they want the character to turn towards.

In order to provide immersion, the player can plug in an Oculus Rift. The Oculus Rift acts like the mouse in the FPS controls or like the right stick in the relative turns controls, that is when the player moves his head, the main character's head also moves in exactly the same manner. This vastly increases the realism and immersion in the game.

It is important to notice that unfortunately the Oculus Rift requires a cable, which can turn into an issue if the player turns a lot in one direction. The cable could get winded and form a knot, or even be unplugged from the computer, making the player lose control.

Graphical User Interface

Since we first target blind people, the graphical user interface (GUI) might not look very important, but we decided to still make it available for the player, in case the player isn't blind at all, or especially when the user plays next to someone that has a good eyesight, and can provide indications in case the game is too difficult.

Since the objectives are explained within the game story, we provided subtitles so that the person helping the player can follow the story, and thus tell what the player is supposed to do. We also set up some displays of some interactions: skip conversation, start test conversation, reload level, open setting settings and zoom.

We also display a timer, so that the person that accompanies the user can, during the next game, compare improvements of the player.

To control the settings in the game, the currently selected option is read and explained to the user in spoken text. This makes it possible for a blind person to navigate the menu. The settings are also shown on the screen so a person that can see it, can select the best options for the current situation more easily.

Our goal with the user interface was that any user, either visually impaired on not, should be able to select his/her desired setting and play the game. A second goal was that someone who is next to the player of the game is able to follow what is going on and can help when needed. This means that what's on screen should look understandable and reasonably stylish.

Code structure

In order for our team to work together simultaneously on the same project, we created a Github project where all of us were working. We worked on a common scene, but were separately working on our parts of the game by creating our own working scenes.

Unity itself imposes a certain code structure. In Unity, all objects that are in the scene are so-called GameObjects. The behaviour of these objects is determined by Components, which can be predefined Unity components, such as a mesh renderer or a physics collider, or it can be user defined scripts.

Quest System

In order to edit story and behaviour of it in the Unity editor, we created a quest system. In the quest system, all variables defining a quest, such a the locations of objects to collect, or which sounds to

player, are stored in a QuestDefinition, which can be used as a Unity component. This makes it possible to edit it in the Unity's graphical editor. The QuestDefinitions can, in turn, create a Quest object, that manages the actual progress of the quest when the game is playing. The Quest object can listen to events thrown by managers and game objects in order to monitor the progress of the player. Quests can also contain subquests, which should either be completed in serial, or in any order the player wants (parallel). Subquests can be made in the Unity scene editor by creating a GameObject containing a SerialQuestDefinition or ParallelQuestDefinition component. This GameObject should contain child GameObjects that contain the quests to complete as part of this parallel- or serial quest. The QuestManager is used to define what the main quest is. It also tracks all the active quests, so that they can easily be influenced, for example to quickly complete them as a cheat.

Some important examples of quests that were used in the game are the:

- (StepWise)FollowWaypointQuest: requires the player to follow Lucy, who appears at several checkpoints in some order. It can also keep track of mistakes the player makes, such as walking in the wrong direction, being hit by something dangerous, or taking very long to reach the next checkpoint. All these errors can be used as triggers for conversations.
- ConversationQuest: Plays a conversation. The quest is completed when the conversation is finished.
- TeleportQuest: Used to teleport the player (and other characters) to different levels.

Note that for the first version of the game, we used a different system, based on a state machine that was defined in code, instead of in the Unity scene editor. The properties of the states could be changed in the scene editor, but the order of the states could not. This turned out to be limiting, and therefore we made the new quest system.

Managers

The game makes use of several singleton classes, called Managers, that can be called from any script and provides functionality that is used often in the game. There are two main types of managers, the settings managers, and the game managers.

Settings Managers

Setting managers make sure that any settings the player chooses will be applied to the entire game. There are four settings, and each one has it's own settings manager:

- Blind Mode Manager handles the visual aids settings. When visual aids are turned off, a black quad is drawn in front of all other graphics.
- The Controls Manager makes sure the correct control scheme is applied. Controls schemes are classes that implement the BaseControls interface. This interface requires the class to return the movement and rotation that should be applied to the character, based on the current user input.

- The Sound System Manager is used to switch between different sound libraries (e.g. Unity default, Phonon 3D and AstoundSound RTI). When the sound system is changed, this managers changes the listener and sources in the game accordingly. It also aids the Audio Manager (see below) when creating audio sources. This makes sure that any audio sources in the game will be processed by the correct sound library.
- The Camera Manager is used to handle the Oculus Rift. It does so by making sure the right camera's are active in the scene, and enabling the Oculus Rift plugin. The controls manager makes sure that the player rotates according to what the oculus rift suggests.

Game Managers

The most important managers are:

- The Audio Manager allows to create audio sources at any desired position in the game world. The audio manager can be used by sending an AudioObject, which contains information on which sound to play, where to play it, and what settings to use (volume, pitch, loop). The manager will in turn return an AudioPlayer object, that other classes can use to control the audio. Audio sources created by this manager will have the current sound setting applied to them, and will automatically be destroyed when the sound is finished playing.
- The Checkpoint Manager keeps track of the last CheckPoint the player has reached. Objects in the game that kill the player can call the checkpoint manager to return the player to the last checkpoint it reached. This works by adding GameObjects containing triggers and a Checkpoint script. This Checkpoint script calls a method on the CheckpointManager when a player enters the trigger of this checkpoint, informing it that checkpoint is reached. The Edge Managers
- The Timer Manager keeps track of the passed time and shows it on the screen when visual aids are turned on. It can be used to keep high-scores. It also writes a log as a csv file containing details on the last playthrough, such as when the player died, and at what time each quest is completed. This can be used for research purposes now, but might be used for high-scores later.
- The Ambient Manager manages ambient sounds. It uses AmbientArea objects that describe in what portion of space the sound should be heard at full volume. If the player moves outside this area, the volume of the sound is linearly interpolated over a predefined distance, so that when you move that distance away from the ambient sound, it cannot be heard. This mimics occlusion and provides a more realistic soundscape.
- The Conversation Manager is used to load and play conversations. A csv resource file contains all conversation, will full transcriptions and conversation IDs for the entire game. The conversation manager loads this CSV file, and can then play any conversation based on the conversation ID. The sound files themselves are named according to this ID as well. This allows for changing the text and audio files of the conversation, without having to change the rest of the game. The ConversationPlayer class manages the playing of the conversation, and makes sure that the correct methods in the Subtitles- and Indicator manager are called.

- The Subtitles Manager is used to show the conversation as subtitles when visual aids are turned on. It is mainly called by the ConversationPlayer, and shows the text of the last spoken section.
- The Indicator Manager is used to indicate when certain game objects are emitting a sound, such as who is talking, and whether Lucy's ringing her bell. This is done by showing an animation in screenspace around the character that is talking.

Other managers are the Pause Manager (stops the game from moving when pause is activated), the Pool Manager (Manage the object pool, making it possible to reuse objects. This saves time because the objects don't need to be created and destroyed) and the Edge Manager (keeps track of whether the player is located near the edge of a level, it uses colliders similar to the checkpoint manager).

Mine/Frog field

The frogs in the swamp in the third level of the game are placed randomly on a grid. In order to make sure that there is always a path from the game, a simple implementation of A* pathfinding is used to make sure that a path exists. From the start position of the grid all cells that are four-connected to it are marked as reachable. Then the same is done from all reachable cells, until no more cells can be marked. If the end position in the grid is marked as reachable the mine layout is good. Otherwise, another random layout is generated and the process is restarted.

Future work

Currently the user can only choose one level of difficulty. This will limit the replayability, and some of the user tests confirmed, and also makes it harder to adapt the game to different players. Indeed, some blind kids are really talented while other struggle much more. Implementing several difficulty levels would mean different modifications such as:

- Increasing the volume of the sounds the player has to focus on, while decreasing that of the noise.
- Reducing the number of obstacles, such as the spirits in the level 1, the number of frogs in level 3.
- Having less noisy machines above the bridge in level 1, of making sure that the machines that aren't relevant for the current section of the bridge cannot be heard.
- Decreasing the amount of time between occurrences of the hooting sound from Boris the owl in level 2.
- Put the waypoints closer to one another. This could be done rather easily by changing the scale of the level while conserving the size of the characters and the volume of the sound.
- Add a button to force noise to be emitted from the destination.
- Add a button that starts a new conversation where Lucy explains again the objective, in the clearest way possible.

Another option we would like to implement is to use a simpler head-tracking device than the Oculus Rift. Even though Oculus Rift provides excellent head tracking, and has good support in Unity, it is quite expensive and requires a cable. Options to replace the Oculus are using the accelerometer and compass of a smartphone, or using image recognition on a webcam. However, both these options are not trivial to implement.

We would also like to implement a version of Legend Of Iris for smartphones and/or tablets, but that would require us to face several challenges. The user interaction would need to be rewritten because there are no arrow keys or controllers available on a handheld device. The interaction could be easy to implement if we represent one stick on the screen for single handed controls, or two sticks on the screen for FPS controls. Another possibility would be to use the gyroscope of the phone to orient the character, the same way it is done with Oculus Rift. We might also run into troubles related to sound libraries not working on these devices, possibly due to limited computational power. Head tracking will also be hard.

Implementing echolocation would be a great contribution to our game. We discovered echolocation when talking with blind people, and were very curious about it. It is the ability of humans to detect objects in their environment by sensing echoes from those objects. It works on the same principle as a sonar. The blind person creates a short sound by tapping their canes, stomp their foot, snap their fingers or make click with their mouths. They are then able to extract some information on the layout and materials of the room they are in based on the sound waves being reflected on nearby objects. Having this in the game can help train blind people in using this skill. Therefore, it goes without saying that we would love to implement this in our game, but many complicated issues would have to be solved. First of all, we would need a drastically more precise sound engine. Since no such engine appears to be available, we would need to implement our own. Based on what is available (GSound for example), it seems reasonable that such an engine would be require much computational power, and it might be extremely hard to reach interactive framerates. Secondly, the way the sound is reflected by an object would also depend of the type of the object, thus, we would need to add some tags to each object of the scene, and then in our sound engine change the reflection according to it.
We already considered implementing echolocation once we got more informations about it, but it quickly appeared as too much work for the amount of time we disposed of.

Conclusion

Making a game for blind people turned out to be a quite challenging but fun task technically. We needed flexible and realistic sound engine, which required quite some research and attempts before we found the engine that seemed to fit our needs in terms of 3D sound propagation. We also learned a lot about sound in general, and about what is important to make it realistic.
In order to make the story and scene in a flexible manner, a lot of coding was needed. The managers, quest system, and other game controller became quite complex in the end, registering many events and triggering the correct actions for each of them, while still maintaining the flexibility required to build a good, fun and useful game.