Pulse

By Philip Smith and Ross Fifield

Table of Contents

Conceptualization and Ideation **3**
Process 4
Design and Technical Implementation **5**
Design 5
Technical Implementation 8
Critical Reflection and Evaluation **12**
Choice of controller 12
Game type 13
Product evaluation 13
Potential commercialisation 13
Youtube URL **14**
Github Repository **14**
Download Link **14**

Conceptualization and Ideation

The conception point for this project was to consider a range of graphical and visualisation techniques which may be suitable in a game development environment. A range of ideas were considered, such as implementation of Voxels for terrain generation, the use of AR mobile phone technology and GPS applied in a pervasive gaming environment, but initial discussions quickly settled on the visualisation of echolocation as a game mechanic. Such an idea is hardly new and has successfully been implemented in modern creative industry media such as Marvel's Daredevil or Christopher Nolan's Batman adaptation of The Dark Knight.

Translating such techniques from broadcast media into interactive media may provide a rich and enjoyable aesthetic and offer relatively unexplored technical implementations of gameplay.

Ideation focused on defining and choosing between a selection of potential prototype projects surrounding echolocation technology. The functional aspects of such a technique could be utilised in a variety of ludic contexts, as shown in Figure 1.

Figure 1. Initial Ideation

Process

The team had a relatively short window in which to produce a functional prototype and so key milestones and objectives were programmed over a week-long project plan, as outlined in Figure 2. The workflow methodology used was MoSCoW (a priority planning technique dividing all work into four categories: Must-have, Should-have, Could-have and Won't-have).

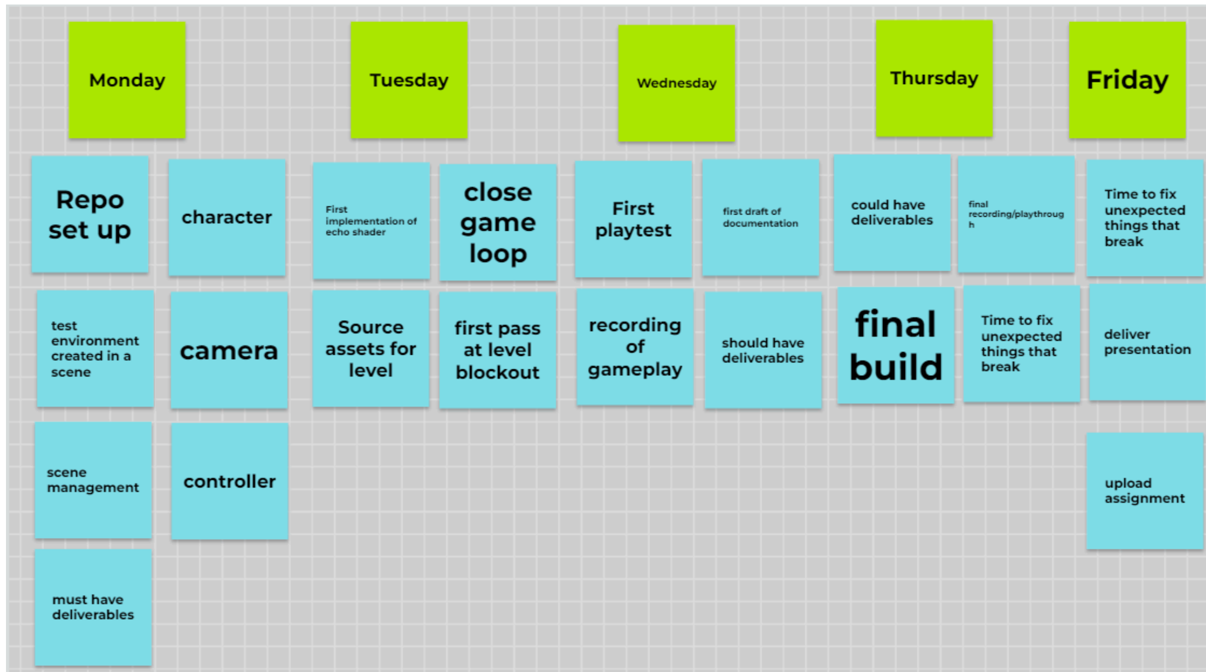
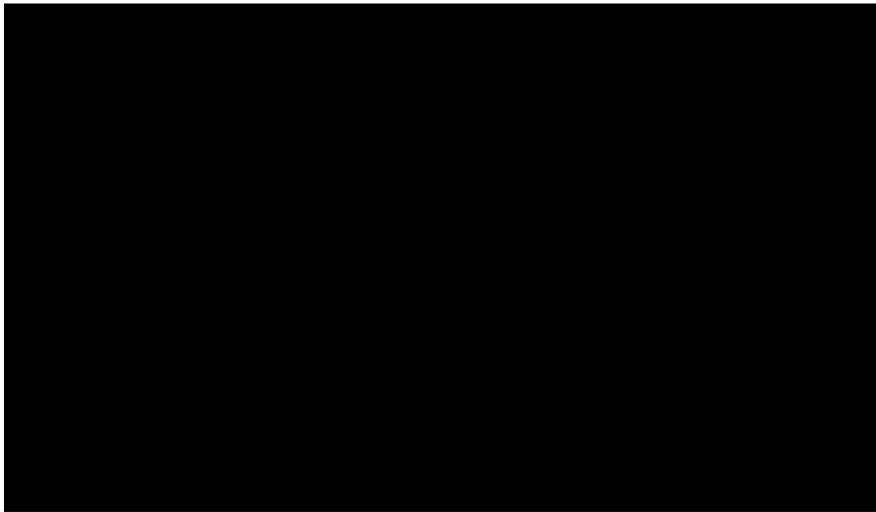


Figure 2. Project Plan

Design and Technical Implementation

Design

Working within the scope of the project, the team decided to focus on the sonar mechanic as a navigation tool. Due to the decision to make every visual object in the scene pitch black without the sonar effect, the sonar mechanic is an essential tool for navigating the level:



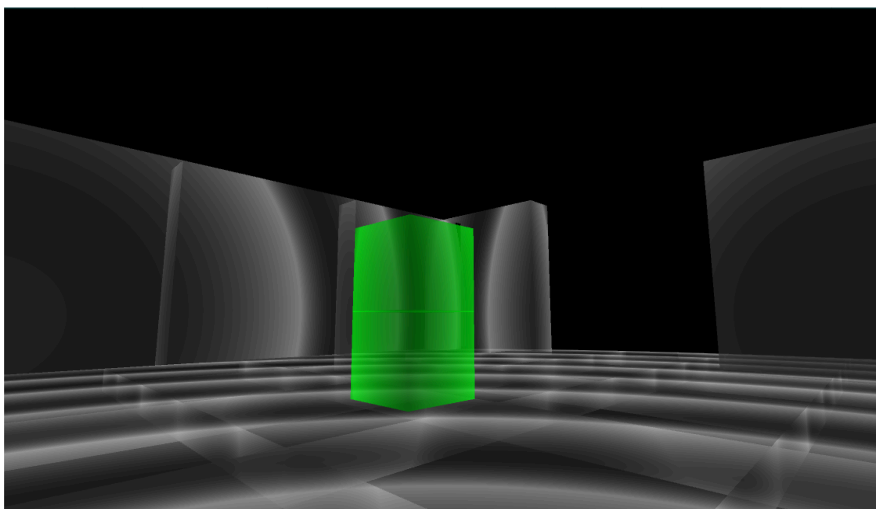
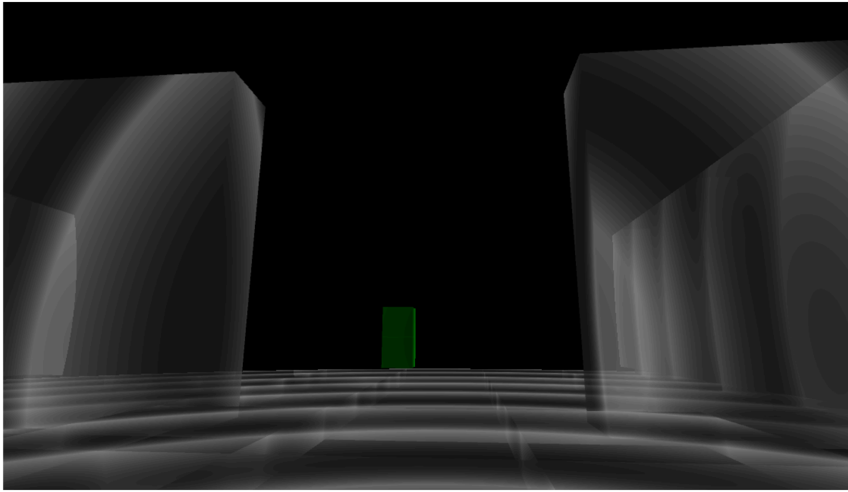


Figure 3. and 4. Before and After sonar is activated

The sonar applies a semi-transparent effect on the objects it interacts with, meaning that by using it, the player can not only see around them, but through objects next to them.



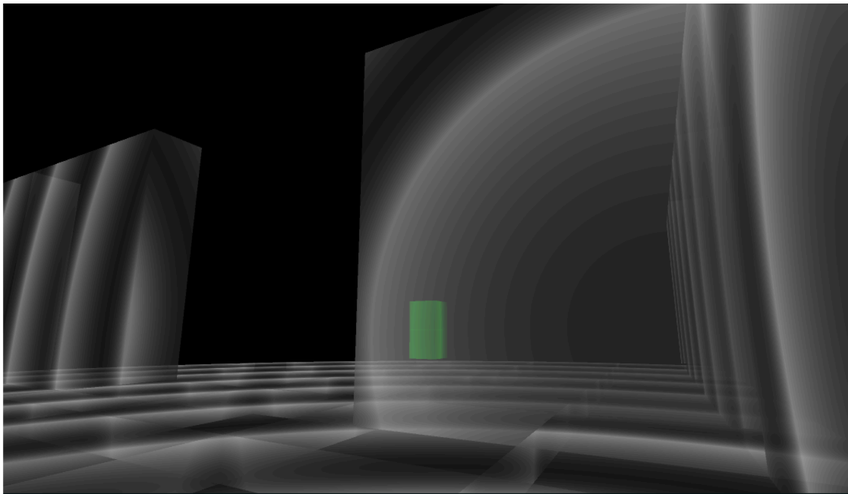


Figure 5. and 6. Showcasing the transparency of objects under the sonar effect

Objects of different purposes show up with different colours under the sonar. 4 different types of objects are accounted for to be graphically distinct using colour indicators: FOOD, Tools, Danger, and Environment. While the demo does not have all 4 present, the initial testing stages demonstrated the effects of Tools, Danger and Environment, with FOOD and Environment being included in the final demo.

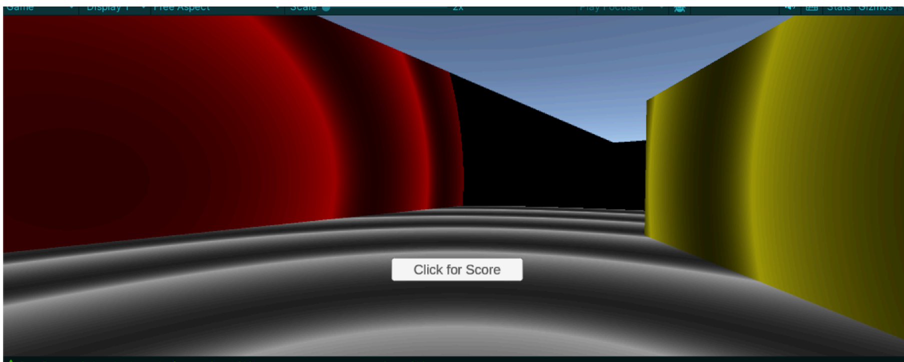


Figure 7. Test Scene demonstrating 3 colors being used for different objects

There are three different means of use available for the sonar. The first is the local pulse, which sends out a sonar effect that reveals the geometry of the environment in a limited radius around the player.

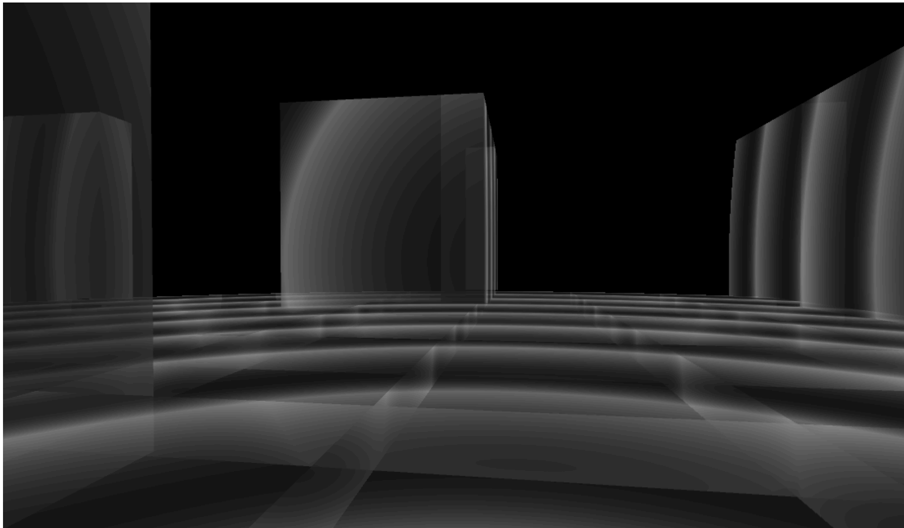


Figure 8. Local Sonar Effect

The second effect is a ranged pulse that is sent out based on where the player's mouse is placed. When it hits an object, the sonar pulse is instantiated on the object's surface.

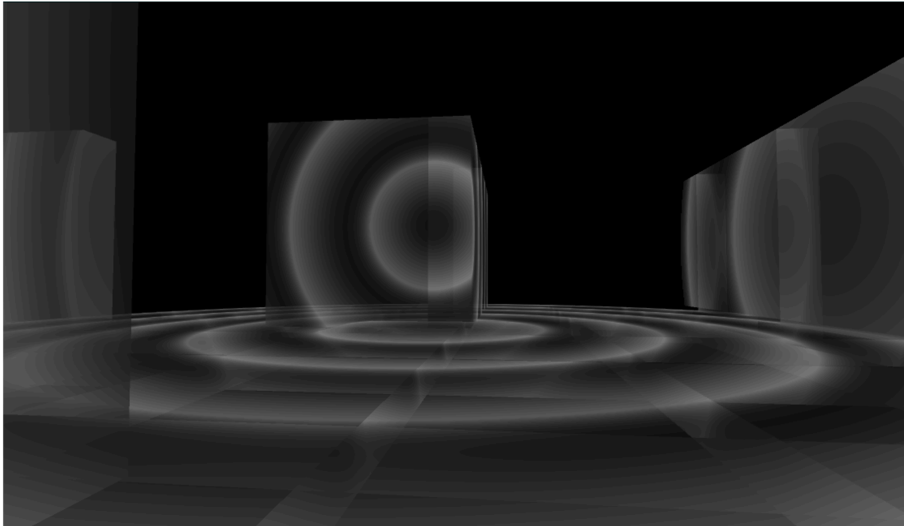


Figure 9. Ranged Sonar Effect

The last action is the sonar recap, which reactivates sonars at each coordinate the player activated their sonar beforehand. The game keeps track of up to 25 sonars, 24 if not including the current new sonar effect.

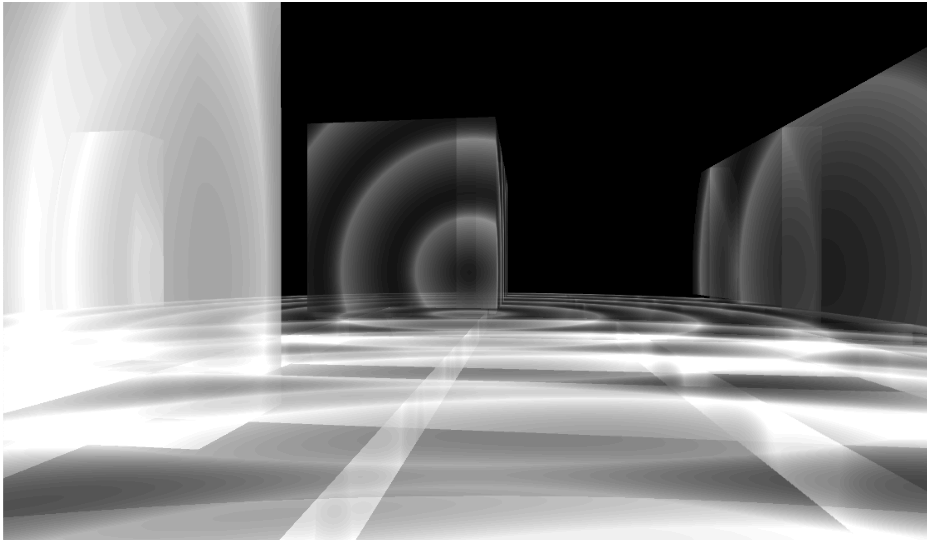


Figure 10. Sonar Recap Effect, with all previous sonars going off at once

Technical Implementation

The graphical effect of the sonar mechanic is built off of the foundation of a Unity package created by Keijiro Takahashi (https://github.com/keijiro/SonarFx). In this package, Takahashi created a shader that mimics the effects of sonar waves. It came with two modes representing two different sonar effects. One simulated a constant directional sequence of waves, all going in a singular direction. The other was spherical, simulating a sonar wave that emanated out from a single point. Our implementation focused solely on the spherical instance. In the original code, the shader was applied as a replacement shader for the whole scene using the camera through the C# script SonarFx.cs. This meant that every object in the scene, regardless of previously declared properties, would adopt the sonar shader:

```
void OnEnable()
{
    GetComponent<Camera>().SetReplacementShader(shader, null);
    Update();
}
```

Figure 11. OnEnable function in original SonarFx.cs script

Additionally, in the original implementation, SonarFx.cs focused on manipulating variables at the shader level:

```
    void Update()
    {
        Shader.SetGlobalColor(baseColorID, _baseColor);
        Shader.SetGlobalColor(waveColorID, _waveColor);
        Shader.SetGlobalColor(addColorID, _addColor);

        var param = new Vector4(_waveAmplitude, _waveExponent, _waveInterval, _waveSpeed);
        Shader.SetGlobalVector(waveParamsID, param);

        if (_mode == SonarMode.Directional)
        {
            Shader.DisableKeyword("SONAR_SPHERICAL");
            Shader.SetGlobalVector(waveVectorID, _direction.normalized);
        }
        else
        {
            Shader.EnableKeyword("SONAR_SPHERICAL");
            Shader.SetGlobalVector(waveVectorID, _origin);
        }
    }
}
```

Figure 12. Original Update() function from SonarFx.cs

This had to change for the purposes of the game's intended design, as it was required for the graphical effect to change based on the type of object it was applied to and where it was instantiated from. Therefore, it was decided to replace the functionality of manipulating the Shader elements directly with a functionality manipulating the same elements through the available materials in the scene:

```
void Update()
{
    var param = new Vector4(_waveAmplitude, _waveExponent, _waveInterval, _waveSpeed);

    for (int i = 0; i < _materialArray.Length; i++)
    {
        _materialArray[i].SetColor("_SonarBaseColor", _baseColor);
        _materialArray[i].SetColor("_SonarAddColor", _addColor);
        _materialArray[i].SetVector("_SonarWaveParams", param);

        if (_mode == SonarMode.Directional)
        {
            Shader.DisableKeyword("SONAR_SPHERICAL");

            _materialArray[i].SetVector
                ("_SonarWaveVector", _direction.normalized);
        }
        else
        {
            Shader.EnableKeyword("SONAR_SPHERICAL");

            _materialArray[i].SetVectorArray
                ("_SonarWaveVectorArray", _originArray);

            _materialArray[i].SetInt
                ("_SonarArraySize", GetOriginArraySize());
        }

    }
}
```

Figure 13. New Update() function for Pulse, altering shader attributes on the Material level

The significant new elements added include the variable _originArray, which passes a vector4 array containing all of the 3D coordinates from where a sonar had been called to

activate, and the function GetOriginArraySize(), which returns an integer that represents how many of those sonar locations in _originArray should be activated at a given time. By using a 4-dimensional vector to pass in 3-dimensional information, we were able to use the 4th element as a simple indicator to the program of whether a sonar effect should be activated at said position. Furthermore, the shader is no longer automatically applied to every object. This leaves flexibility for future development in case some objects require slightly different or completely different shaders.

Outside of the altered SonarFx.cs, there is the completely new script EchoManager.cs. This script manages the 3D objects in the scene that are affected by the sonar effect and determines how they are affected, the latter referring to what colour they show up as under the sonar waves. The function Echo()carries out this functionality, going through 4 different lists of objects generated from the scene based on their tags: FOOD, Tools, Danger, and Environment:

```
for (int i = 0; i < FOOD.Length; i++)
{
    FOOD[i].GetComponent<Renderer>().material.SetColor("_SonarWaveColor", _foodColor);

    FOOD[i].GetComponent<Renderer>().material.SetVectorArray(
            "_SonarWaveVectorArray", sonar.originArray);

    FOOD[i].GetComponent<Renderer>().material.SetInt
                ("_SonarArraySize", sonar.GetOriginArraySize());

    FOOD[i].GetComponent<Renderer>().material.SetInt
                ("_Radius", thisRadius);

    FOODMaterialColors[i] = FOOD[i].GetComponent<Renderer>().material.
        GetColor("_SonarWaveColor");

}
```

Figure 14. FOOD loop instance in Echo() that changes the attributes of all objects in the scene classified as ‘FOOD’, and stores the colour of the material to create a fading effect later in the code

Additionally, EchoManager instantiates and calls SonarFx.cs to store the sonar activation coordinates on _originArray for future use once the sonar effect concludes its life cycle:

```
_sonarOn = false;

Vector4 carry = Vector4.zero; // Store previous vector in the array for appending
bool appending = false;

for (int i = 0; i < sonar.originArray.Length; i++)
{
    if (sonar.originArray[i].w == 1)
    {
        sonar.originArray[i].w = 0;

        if (i == 0)
        {
            appending = true; // New sonar was created. Append it to the array
        }
    }

    if (appending)
    {
        Vector4 temp = sonar.originArray[i];
        sonar.originArray[i] = carry;

        carry = temp;
    }
}
```

Figure 15. Once the sonar effect ends, the coordinate vector of the current sonar effect is appended to the array at the second slot, moving all other stored coordinates down by one, deleting the vector in the last available slot

Finally, there is PublicSonarShader, a public-access, altered version of the hidden shader file provided by Takahashi's package. While the Math Takahashi used in the shader was not altered in order to retain the sonar effect, a few additions were implemented for the sake of the tech demo's purpose. Firstly, the elements mentioned before, _originArray and its size, play vital roles in determining where and how many sonar effects occur.

```
for (int i = 0; i < _SonarArraySize; i++){

    if(_SonarWaveVectorArray[i].w == 1) {

        float dist = length(IN.worldPos - _SonarWaveVectorArray[i].xyz);
```

Figure 16. For every 4D vector in the array according to _SonarArraySize, check that they are tagged as active (4th-element is 1) and then find the distance between the fragment

Then an additional variable passed in, _Radius, determines how far the effect travels out from the center:

```
if (dist > _Radius){

    w = 0;

}
```

Figure 17. Set w to 0 if the distance between the point in the world and the sonar origin is too far based on the Radius

Finally, to create the translucent x-ray effect, the Sub Shader properties were altered to allow for transparency:

```
//Tags { "RenderType" = "Opaque" }

Tags{ "RenderType"="Transparent" "Queue"="Transparent"}

Blend SrcAlpha OneMinusSrcAlpha

ZWrite Off
```

Figure 18. Settings necessary for transparency applied in the Sub-Shader

And the shader determines that if the Emission (which represents the hues of the wave) equates to a magnitude larger than the 0 vector, then to set the alpha of the surface to 0.7, giving it a semi-transparent effect. Else, the shader becomes opaque again at alpha 1.

```
// Apply to the surface.
OUT.Emission += _SonarWaveColor * w + _SonarAddColor;
OUT.Albedo = _SonarBaseColor;

if (OUT.Emission.x > 0 ||
    OUT.Emission.y > 0 ||
    OUT.Emission.z > 0)
{
    OUT.Alpha = 0.7;
}

else{
    OUT.Alpha = 1;
}
```

Figure 19. If either of Emission’s elements are over 0, apply transparency. If not, make the material opaque.

Critical Reflection and Evaluation

Choice of controller

The first task was to implement the 3 C’s - Camera, Character and Controller. Initial discussions could have led the game type and consequent application of technology in a variety of directions. An early idea was to consider the use of echolocation as a navigation tool and so we chose to implement a physics based controller. This turned out to be unnecessary and created some bugs. The player could for example over rotate and in some cases clip through scenery entirely. Such issues did result in us considering how we were going to apply echolocation. It could be used for perception, navigation, as a detection tool and in a sound based game, perhaps even as a given functional use in interacting with terrain, scenery and other agents.

Game type

The application of this technology is of course one of the big issues surrounding its use. A variety of different game genres were considered and we quickly pivoted from an open, platformer style game to an enclosed maze style game. We felt that the technology could be best used as a detection and navigation tool and so our implementation choices started pivoting towards this kind of game as a result.

This is reflected in the inclusion of ‘enemy’ and ‘food’ types of objects, which would in turn return different kinds of echos and result as visual cues for gameplay. One entertaining idea we considered is that this technology could be utilised in both ‘push’ and ‘pull’ types of games, which is perhaps somewhat novel. It could just as easily appear in a stealth game where avoidance and stealth were key elements of gameplay, as it could appear in ‘hunting’ style games, where the player had to hunt down enemies in a maze. Our feeling was that this multi-functional usage should not be disregarded in future ideation.

Product evaluation

As mentioned previously, the player controller was initially designed as a physics based control, in order to facilitate emergent and divergent styles of movement in play, but midway through the project, was changed to a simpler input style controller system.

The environment was produced as a non-linear interconnecting maze, with lots of interconnecting corridors and opportunities to traverse around the area. Dead ends and backtracking were avoided, in order to facilitate hunting and evasion style gameplay. Given that visual cues in the level were limited to activation of echo-location, it was felt unnecessary to add extra features such as color, lighting and physical assets such as furniture and other obstacles.

Any evaluation must reasonably ask the question - did the product do what we expected it to? Given the scope and timescales we were operating within, broadly speaking, we can assert that as a proof of technical concept it works. It is possible to perceive and traverse the environment and there are interesting opportunities to incorporate playful interaction among a variety of games.

Potential commercialisation

This product could be commercialised via a creative media licence from companies such as Marvel or Disney. The features demonstrated may also be of interest to publishers who seek investment in experiential and horror titles and may be of more wider interest as a feature for games which rely heavily on perception and navigation as key features of play.

Youtube URL

https://youtu.be/My_f3NBkEeE

Github Repository

https://github.com/RossFifield/Pulse

Download Link

https://rafifield.itch.io/pulse