Indice:

1. Justificación.
2. Agradecimientos.
3. Resumen.
4. Objetivos
5. Diseño del juego.
6. Desarrollo del juego.
7. Sonido:

* Sonido 2D. (posicionado en el espacio)
* Sonido Visual (representación visual del sonido)

1. Herramientas.

* Visual Studio.
* SFML.
* Box2D.
* Tiled.
* TinyXML.
* Gimp.
* Audacity.
* GitHub y SourceTree

1. Mecanicas.

* Sonidos como mecánica.
* Caminar.
* Pinchos
* Enemigos.
* Sigilo.
* Mecanismos y puertas.
* Rocas.
* Agua.

1. Diseño de niveles.
2. Físicas.

* Mascaras de colisiones.

1. Motor Visual.

* Sprites
* VertexArray y TriangleStrips

1. Patrones de diseño.

* RAII.
* Façade.
* Abstract Factory.

1. Lenguaje.

* C++11

1. Metodologia.
2. Justificación.
3. Agradecimientos.
4. Resumen.

Echoes es un juego de terror que utiliza una formula sencilla: eliminar la visión y agudizar el oído. Consiste en recorrer una serie de mapas formados por salas totalmente a oscuras, para ello tendremos que guiarnos únicamente por el sonido de nuestros pasos, buscando la salida sin morir en el intento. Suena fácil, pero estaremos rodeados de peligros y cada mapa será más complicado que el anterior.

1. Objetivos.

* Mejorar la capacidad de auto aprendizaje.
* Aprender a realizar un proyecto de tamaño medio.
* Unificar diferentes tecnologías para realizar el proyecto.
* Realizar un videojuego desde cero.
* Superar los problemas que surgen a lo largo de la creación de un proyecto.

1. Diseño del juego.

Solamente seremos capaces de situar visualmente nuestros pies, que serán unas huellas blancas en la pantalla totalmente negra, esto será así mientras no nos movamos, porque el sonido de nuestros pasos, o cualquier otro sonido, generará una serie de ondas visuales que nos permitirán vislumbrar la forma del mapa y los peligros que nos rodean, rebotando en las paredes y cambiando de color dependiendo de la zona por la que pasen.

Estas ondas sonoras serán más intensas y duraderas cuanto más fuerte sea el sonido producido, y a lo largo de los mapas se irán añadiendo mecánicas para generar más sonido, o menos, según nos convenga, lo que da una profundidad y posibilidad de creación de puzles infinita.

1. Desarrollo del juego.

1. Diagrama de clases.
2. Sonido:

* Sonido 2D.
* Sonido Visual

1. Herramientas:

* Visual Studio.

Para la realización de este proyecto se ha optado por utilizar el entorno de desarrollo Visual Studio 2015 puesto que después de haber probado diferentes opciones a lo largo de la carrera, como son Codeblocks o eclipse, lo considero muy superior en calidad de las herramientas de depuración y ayuda a la hora de escribir código mas rápidamente.

* SFML.

Como motor gráfico se ha usado SFML por considerarlo una herramienta excelente que aporta tanto herramientas como tutoriales y ayudas para usarlas, es estupendo a la hora de utilizar gráficos en 2D y aporta librerías de audio, eventos y control de tiempo.

* Box2D.

Puesto que es un juego que requería de una componente física muy potente, con manejo de mascaras de colisiones, sistema de partículas y resolución de colisiones complejas y rápidas se ha apostado por la librería de físicas Box2D, a la que se le ha dotado de una fachada con el patrón façade para poder sustituir la librería de manera rápida en caso de necesitarlo.

* Tiled.

Para la realización de los diferentes mapas del juego se ha utilizado la herramienta tiled, de la cual se han utilizado únicamente las capas de objetos, esta herramienta te permite dibujar en una rejilla la forma del mapa deseado utilizando capas para distinguir entre los diferentes objetos y guarda la información en un fichero de texto XML, el cual leeremos de la manera que nos convenga desde la aplicación y podremos construir nuestros mapas.

* TinyXML.

Como ya se ha comentado, al utilizar la herramienta Tiled para crear los mapas se requiere de una librería de lectura de ficheros XML, y para ello se ha utilizado tinyXML, que permite recorrer un fichero en este formato a través de las etiquetas de manera fácil e intuitiva.

* Gimp.

Para la realización del sprite de los pies del personaje se ha utilizado la herramienta Gimp, que es un potente entorno de diseño gráfico gratuito.

* Audacity.

Con Audacity se han modificado y ajustado los audios capturados para el juego para adaptarlos a las necesidades que teníamos al realizar un juego de terror, como ecos o reverberaciones.

* GitHub y SourceTree

Como repositorio y sistema de control de versiones se ha utilizado una cuenta en GitHub y el entorno grafico SourceTree de Atlassian.

1. Mecanicas:

* Sonidos como mecánica.

El sonido será la principal fuente de información de lo que sucede en el mundo que nos rodea, este será representado con líneas rectas cuya velocidad, intensidad y duración dependerán del la fuerza y volumen del sonido que la produce. Esto dará como resultado una pantalla cargada de información a través de estas ondas que rebotando contra los límites del mapa nos darán información de hacia donde tenemos que ir y que nos rodea, ya que cuando una de estas ondas pase sobre los diferentes objetos del juego (palancas, pinchos, etc) se colorearan durante el periodo en el que están situadas sobre el objeto de un color característico del objeto, por ejemplo, la salida de cada mapa se coloreara de verde.

* Caminar.

El simple hecho de caminar producirá ruido a causa de nuestros pasos, estos generarán una buena cantidad de ondas sonoras que nos harán de guía en nuestro camino.

* Pinchos

Tendremos que tener cuidado de hacia dónde nos dirigimos, ya que la mazmorra estará llena de pinchos (rojo) que tendremos que sortear.

* Enemigos.

A lo largo y ancho de los mapas nos iremos encontrando con unos peculiares enemigos, de los cuales no conoceremos su aspecto puesto que no los vemos, pero si les escuchamos, y ellos a nosotros, ¡tened cuidado!

* Sigilo.

A veces caminar es demasiado peligroso si nos acechan enemigos, para esto tenemos la capacidad de avanzar sigilosamente y sin hacer casi ruido, lo malo es eso mismo, ¡que sin ruido no vemos hacia donde estamos yendo!

* Mecanismos y puertas.

A lo largo de nuestro camino nos encontraremos con una serie de mecanismos (color amarillo) que al pisarlos nos darán acceso a nuevas zonas antes inaccesibles.

* Rocas.

Cuando una zona está vigilada por un terrible enemigo tendremos la posibilidad de lanzar una piedra para distraerle, puesto que siempre van a olfatear el lugar del cual les llega un ruido.

* Agua.

El agua es un enemigo “silencioso”, no nos matará, pero nos hará avanzar más despacio y hacer más ruido, tened mucho cuidado con ella.

1. Diseño de niveles:
2. Físicas:

* Mascaras de colisiones.

Dada la complejidad de las colisiones y la gran variedad de objetos con diferentes respuestas a las colisiones dependiendo que tipo de objeto choca con cada uno de ellos, se ha implementado un sistema de mascaras de colisiones para indicar a box2d que colisiones son sensores, cuales producen una colisión real y que resultado tiene cada una de las combinaciones.

1. Motor Visual:

* VertexArray y TriangleStrips

Las ondas que simulan el sonido tenían que ser líneas continuas de un tamaño y grosor variable cuyo color fuera transparentándose a modo de estela y que en ciertas zonas del espacio cambiara de color, este tipo de objeto era nuevo para mí y no sabía cómo podía realizarlo utilizando las herramientas de SFML.

Para una primera aproximación de esta idea utilicé un vector variable de rectángulos, cada pocos milisegundos se generaba un rectángulo en la posición actual de la partícula situada en la cabeza del objeto, este vector se recorría en cada iteración de la aplicación y se otorgaba a cada rectángulo el color y transparencia que tocaba.

Esta solución a pesar de ser funcional, no daba el resultado final que se quería conseguir, además de ser muy costosa en tiempo de ejecución.

Finalmente, y tras una exhaustiva búsqueda de una mejor implementación de esta idea encontré las herramientas VertexArray y TriangleStrips, que juntas me daban la solución perfecta a mi problema.

VertexArray permite almacenar la información de una serie de puntos geográficos, cada uno con su información de color, y triangleStrips permite formar una forma geométrica compleja a partir de un vertexArray, donde cada 3 puntos forma un triángulo cuyo color es un gradiente de los colores de los puntos que lo componen.

Con esta herramienta fui capaz de almacenar cada pocos milisegundos la posición de los vértices delanteros de la cabeza de la onda en un vertexarray, y cada iteración formar un TriangleStrip que generara la forma deseada con los colores deseados, dado que cada punto espacial guarda la información de su propio color.

Solo faltaba saber cuántos vértices guardar en cada momento dependiendo del tamaño de estela deseado.

1. Patrones de diseño:

* RAII.

El patrón RAII hace el código seguro frente a excepciones, se utiliza para liberar recursos antes de permitir a las excepciones que se propaguen (para evitar fugas de memoria) escribiendo destructores apropiados nos ahorramos escribir código de “limpieza” duplicado y disperso por el código fuente entre bloques de manejo de excepciones que pueden ser ejecutados o no.

Este patrón se apoya en la característica de que posteriormente a una [excepción](https://es.wikipedia.org/wiki/Manejo_de_excepciones), el único [código fuente](https://es.wikipedia.org/wiki/C%C3%B3digo_fuente) que con seguridad es ejecutado es el de los [destructores](https://es.wikipedia.org/wiki/Destructor_(inform%C3%A1tica)) de objetos que residen en la [pila](https://es.wikipedia.org/wiki/Pila_(inform%C3%A1tica)).

Ademas permite asegurarse que las clases de nuestra aplicación no crean constructores por defecto, constructores de copia ni operadores de asignación, esto se realiza creando una clase base que inhabilita estos constructores.

* Façade.

El patrón Façade se utiliza para desacoplar las diferentes funcionalidades, librerías o clases de una aplicación o para aportar interfaces simples a estructuras complejas, esto se realiza creando clases e interfaces intermedias que se comunican con la librería que se quiere usar, de esta manera aporta una interfaz de uso a esta parte de la aplicación.

* Abstract Factory.

1. Lenguaje:

* C++11