There’s time

Aram Àvila Salvadó
Pla d’estudis de l’estudiant
TFG – Videojocs – Codi 06.587

Esther Arroyo Garriguez
Jose Luis Cánovas Izquierdo

9 de Juny de 2019

FITXA DEL TREBALL FINAL

Títol del treball:	*There's time*
Nom de l'autor:	*Aram Àvila Salvadó*
Nom del consultor/a:	*Esther Arroyo Garriguez*
Nom del PRA:	*Jose Luis Cánovas Izquierdo*
Data de lliurament (mm/aaaa):	*06/2019*
Titulació o programa:	*Grau de Multimèdia*
Àrea del Treball Final:	*06.587 - TFG-Videojocs*
Idioma del treball:	*Català*
Paraules clau	*Videojoc, Indie, Demo-Tècnica*

Resum del Treball (màxim 250 paraules): *Amb la finalitat, context d'aplicació, metodologia, resultats i conclusions del treball*

El treball consisteix en investigar i, si es pot, desenvolupar un videojoc que exploti el concepte de la informació perfecta, segons el qual en un joc qualsevol tots els jugadors tenen la mateixa i tota la informació relativa l'estat del joc. L'objectiu final és desenvolupar mecàniques o idees que puguin explotar el concepte i permetin crear un videojoc que les apliqui d'una manera clara i entenedora pel jugador.

En la versió final del videojoc, el jugador controlarà directament la seva càmera i es podrà moure per un entorn en quatre dimensions (tres espacials i una temporal). Aquest entorn serà a la vegada, el temps i l'espai pel que es mourà una criatura tridimensional, dues espacials i una temporal.

L'objectiu del joc serà fer arribar tant al jugador com a la criatura al final de cada nivell. La progressió en els nivells n'anirà incrementant tant la dificultat com els elements amb els que tant el jugador com la criatura poden interactuar per completar-lo.

Es desenvoluparà utilitzant C++, OpenGL i les llibreries GLEW, GLUT, GLM principalment. El resultat final serà un executable més els arxius addicionals com poden ser els nivells o les textures.

El resultat no ha estat l'esperat, tot i que s'han pogut explotar moltes de les idees que es podrien haver aplicat al joc.

Abstract (in English, 250 words or less):

This paper has its main focus in investigating and developing, if deemed possible, a videogame that exploits the idea of "Perfect information", a concept by which in a game of any sort, all parties involved have all the information

related to the current state of the game. The final objective will be coding a videogame such as the one described.
The developed videogame consists of a camera that will be directly controlled by the player in a 4 dimensional environment whose 3 spatial dimensions will be width, height and time of a 3 dimensional world. In that world, a creature constrained in 2 spatial dimensions will be indirectly controlled by the player.
The videogame will be divided in multiple levels of increasing difficulty that will progressively include new interactive elements and concepts for both the player and the creature.
The software will be developed in C++, using OpenGL and the libraries GLEW, GLUT and GLM mainly. The end product will be an executable file and additional files such as levels and textures.
The end result has not been the expected product but the initial concept has studied in depth.

Índex

1. Introducció

1.1 Context i justificació del Treball

Investigant sobre la teoria i mecàniques que construeixen un videojoc vaig descobrir el concepte de la informació perfecte; un concepte aplicable a qualsevol joc, sigui digital, de sobretaula o un esport. Aquest concepte diu que un joc tindrà informació perfecte quan tots els jugadors tinguin tota la informació del que passa en la partida. Això significa que en el joc no hi ha d'haver elements d'atzar o amagats tot i que sempre hi haurà la imprevisibilitat de la següent jugada.

En aquesta categoria podem trobar jocs com els escacs, on els dos jugadors saben la posició de totes les peces o el futbol, on els jugadors saben on són tots els altres. En ambdós jocs, la situació actual és coneguda per tots els agents tot i que el futur de la partida segueix sent un misteri. Quin serà el pròxim moviment? Cap on anirà el pròxim xut?

Per altra banda, en jocs com ara el pòquer o el domino, sempre hi ha elements que son fora del nostre coneixement com ara quines cartes o fitxes tenen els altres jugadors o quines cartes queden per sortir. En aquesta mena de jocs no tenim tota la informació relativa a l'estat actual del joc.

En el cas dels videojocs, aquest concepte no ha estat més que el resultat de les mecàniques aplicades, no ha estat mai el tema central al voltant del qual es construeix l'aplicació; era la conseqüència, no la decisió.

En el mercat actual no hi ha cap videojoc que exploti aquesta idea, de manera que s'obre la possibilitat de crear mecàniques mai vistes.

La interactivitat dels videojocs permetrà portar aquest concepte un pas més enllà i buscar el cas de la informació meta-perfecte, en el que el jugador podrà saber tot el que passa en la partida simultàniament. La intenció consisteix en investigar si darrere d'aquesta idea hi ha un videojoc i si hi és, desenvolupar-lo.

1.2 Objectius del Treball

- Estudiar el concepte de la informació perfecte
- Desenvolupar una idea central que permeti explotar-la
- Desenvolupar mecàniques que complementin la idea central
- Aplicar aquestes idees i mecàniques en la realització d'un videojoc

1.3 Enfocament i mètode seguit

Inicialment s'estudiaran els motors de creació de videojocs ja disponibles al mercat, com podrien ser Unity o Unreal Engine. Se n'avaluaran les capacitats, requisits i limitacions i s'estudiarà la possibilitat d'utilitzar-los com a base.

En el cas de que cap d'aquests motors pugui ser utilitzat se'n crearà un utilitzant C++ i OpenGL, ja que no estarà limitat per cap framework.

El fet que el joc es basi en un concepte nou fa que sigui difícil preveure el resultat final de manera que es seguirà un procés de desenvolupament iteratiu. Consistirà en investigar el concepte sobre paper i desenvolupar les primeres mecàniques. A continuació es programarà una demo amb aquestes mecàniques i s'estudiarà si el seu comportament és l'esperat o si cal canviar-les. Un cop les mecàniques s'hagin solidificat es tornarà a la fase d'investigació per mirar quines altres es poden afegir o quin canvis s'haurien de fer per seguir avançant.

1.4 Planificació del Treball

Per realitzar el projecte s'utilitzarà l'IDE Visual Studio. Es farà servir OpenGL i s'utilitzaran les llibreries GLFW com a framework sobre OpenGL, GLM per la gestió de vectors i matrius i FreeType per la renderització de text sobre la pantalla.

La llista de tasques a realitzar és la següent:

1- Plantejament de possibilitats: Estudiar les diferents maneres d'explotar la idea. Temps estimat: 12 dies – Entrega Pac1

2- Inici del procés iteratiu de desenvolupament.

   a. Demo tècnica: Per comprovar les possibilitats del motor triat i decidir si continuar treballant-hi o canviar-lo. Temps estimat 2 dies.

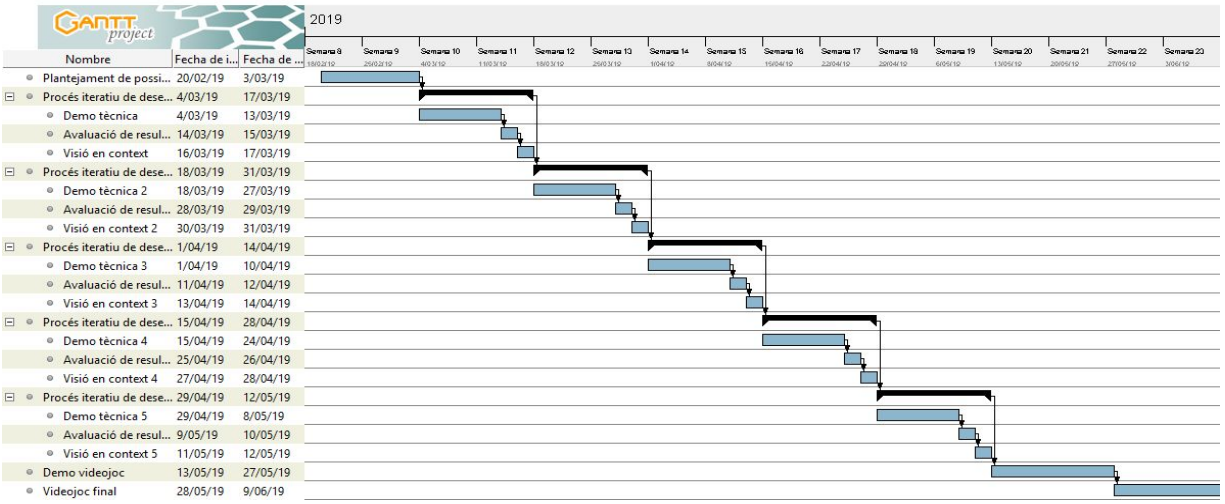
   b. Avaluació de resultats: Comprovar quines de les mecàniques funcionen com és esperat i realitzar els ajustos que siguin necessaris. Temps: 15 dies

   c. Visió en context: Agafar perspectiva sobre la demo realitzada i decidir quines son les següents mecàniques a implementar tenint en compte la feina feta. Temps 2 dies.

Temps total pel procés iteratiu: 2 mesos – La segona iteració serà l'entrega de la PAC 2

3- Demo videojoc: Expandir la demo tècnica per fer la base del videojoc. Valorar si s'han assolit els objectius de desenvolupament.Temps estimat: 2 setmanes – Entrega PAC 3

4- Videojoc final: Un cop s'hagi completat el videojoc es realitzaran testos amb usuaris i es poliran i ajustaran les mecàniques i interaccions. – Entrega PAC Final



1.5 Breu sumari de productes obtinguts

El resultat obtingut no ha estat l'esperat ja que la idea inicial era desenvolupar un videojoc que mostrés tot el potencial del concepte de la informació meta-perfecte. En canvi, el producte obtingut és més semblant a una demostració tècnica de les diferents mecàniques que es podríen aplicar a un hipotètic videojoc com el plantejat inicialment.

Caldria revisar aquestes mecàniques i buscar la manera que tinguin un impacte més important en la relació entre el jugador i el planar de manera que resoldre cada nivell no sigui trivial. En la meva opinió, el resultat final serveix per veure i entendre com funcionarien aquestes mecàniques però en l'estat actual no permeten construir nivells prou interessants per al jugador.

Segons la llista d'objectius inicials:

- Estudiar el concepte de la informació perfecte: Assolit, s'ha estudiat com funciona el concepte en altres jocs i se n'han destil·lat les idees principals.
- Desenvolupar una idea central que permeti explotar-la: Assolit, s'ha desenvolupat la idea de representar el temps del planar com a espai pel jugador.
- Desenvolupar mecàniques que complementin la idea central: Assolit, s'han desenvolupat diverses mecàniques que s'arrelen en la idea central descrita anteriorment.
- Aplicar aquestes idees i mecàniques en la realització d'un videojoc: No assolit. S'ha programat un videojoc però li manquen elements imprescindibles per fer-lo interessant, com ara el risc-recompensa o un sistema de trencaclosques real en cada nivell.

1.6 Breu descripció dels altres capítols de la memòria

1- Tria de tecnologies
2- Idees preconcebudes
3- Primeres mecàniques
4- Desenvolupament de mecàniques
    a. Nivell sòlid
    b. Restricció del jugador a l'entorn físic
    c. Interaccions entre el jugador i el planar
    d. Accions del planar
    e. Distorsions
    f. Plataformes
    g. Zones de mort
    h. Meta
5- Aspectes tècnics
    a. Estructura del codi
    b. Funcionament general i renderitzat
    c. Estructura i càrrega de nivells
    d. Detecció de col·lisions i físiques

2. Resta de capítols

2.1- Tria de tecnologies

La plantejament inicial consistia en utilitzar un motor ja existent per poder centrar els esforços plenament en desenvolupar les mecàniques necessàries. Després d'un període de documentació, a través d'articles sobre diversos motors, estudi d'altres projectes, converses amb coneguts, etc, vaig arribar a la conclusió que era millor programar un motor gràfic de zero.

Els factors que em van dur a aquesta tria van ser el fet que cap dels motors comparats em donava prou llibertat per completar els requisits del projecte. Addicionalment, no he treballat mai amb aquests motors, de manera que tenia per davant un procés d'aprenentatge que no podia estar segur que acabés resultant útil.

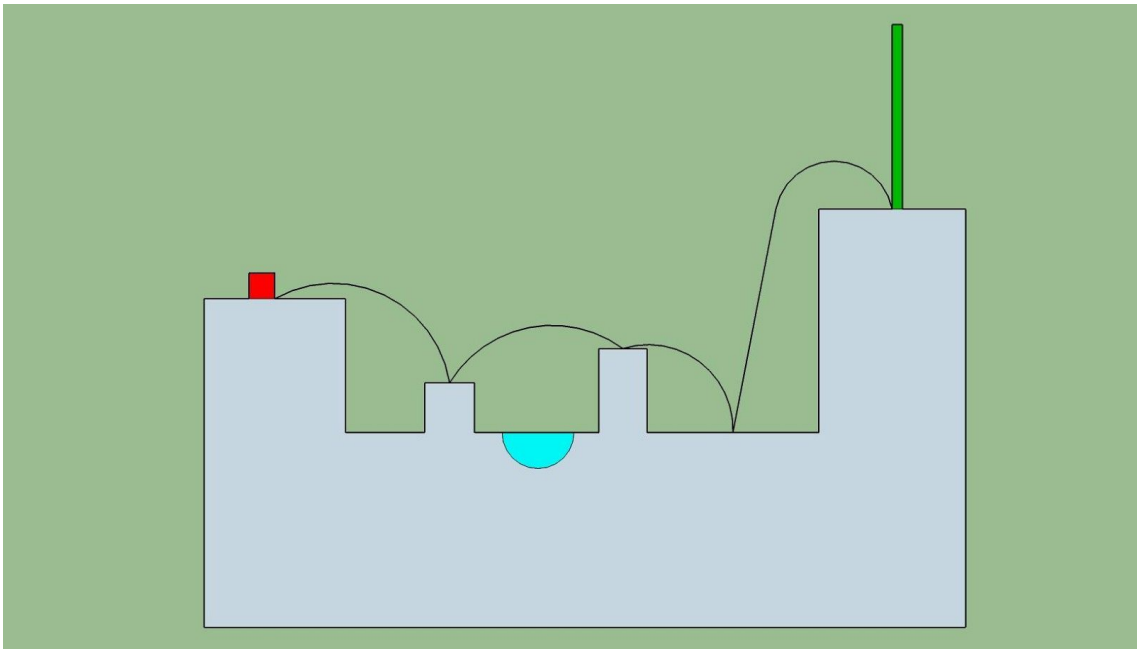
Un cop descartats, la tria va resultar clara: OpenGL. Es tracta d'un motor gràfic molt potent i compatible amb shaders GLSL, que permeten transmetre la càrrega del renderitzat a la GPU. El videojoc en si podria estar programat en diversos llenguatges; Python, C#, Java, C++ o Visual Basic entre altres. Degut al meu domini i experiència amb cada llenguatge vaig reduir la llista a Java i C++. Ja havia realitzat altres projectes i aquests dos llenguatges son bastament populars, de manera que podria trobar els recursos necessàries i resoldre els dubtes que sortissin durant el desenvolupament amb facilitat.

Finalment, la tria del llenguatge C++ va ser principalment perquè Java no pot interactuar directament amb OpenGL, s'ha d'utilitzar una de múltiples llibreries que fan de pont. Això afegeix una altra capa en propi programa que en alenteix el funcionament i des del punt de vista del desenvolupador, afegeix un pas addicional a tenir en compte a l'hora de treballar, cosa que dificulta i complica el progrés del projecte.

Finalment, l'ús de shaders GLSL permet alliberar la CPU de la càrrega de renderitzar l'escena de manera que es pot emetre una quantitat molt més gran d'imatges per segon, fent que el videojoc sigui visualment fluid. Per limitar l'ús que se'n fa, es fixarà el renderitzat a 60 fps.

2.2- Idees preconcebudes

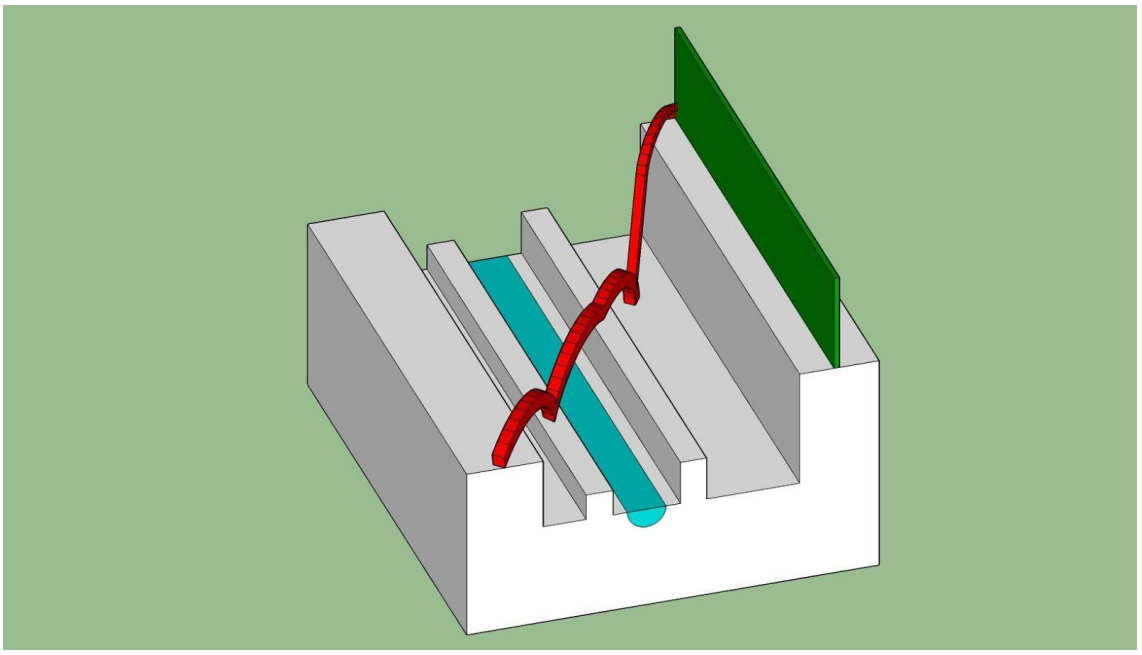
Partim del concepte informació meta-perfecte. L'estructura de l'entorn tindrà 4 dimensions; consistirà el temps, és a dir, el jugador es podrà moure lliurement i percebrà el temps en el joc com el temps real. El jugador es mourà per un sòlid en 3D que estarà compost per l'alçada, amplada i el temps d'un altre entorn amb 2 dimensions espacials i 1 de temporal



Il·lustració 1: Entorn 2D del planar

En la il·lustració 1 podem veure un nivell model. El planar es representat pel quadrat vermell, el sòlid del nivell pel polígon gris i la meta per la barra verda a l'extrem dret. Si es tractés d'un videojoc de plataformes, el planar es mouria cap a l'esquerra fent els salts convenients fins arribar a la meta.

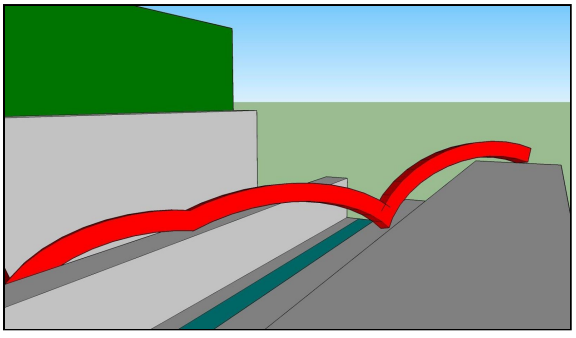
Ara, si agaféssim cada una de les imatges que hem vist a la pantalla mentre jugàvem aquest nivell i les poséssim una darrere de l'altre acabaríem obtenint un sòlid com el següent:



Il·lustració 2: El temps en 2D es profunditat en 3D

En aquesta imatge es veu clarament com l'amplada i l'alçada en la il·lustració 1 segueixen mantenint el mateix rol, mentre que el que era temps mentres "jugàvem" aquell nivell, ara és la profunditat.

Aquest concepte permetria a un jugador moure's pel que seria el temps del planar.



Il·lustració 3: El jugador veu el temps del planar com a espai.

Això ens permetria explotar el concepte de múltiples maneres i a diferents escales:

- Construït un imperi.

Al jugador se li presenta un nivell de grans dimensions que representarà un món complet, que podria ser circular pels planars i cilíndric pel jugador, seguint l'analogia anterior. En aquesta idea, hi haurien múltiples planars que es

mourien i interaccionarien entre ells i amb un entorn dinàmic, susceptible a l'erosió del temps i a les accions dels planars.

L'objectiu del jugador seria aconseguir fer prosperar la societat o civilització dels planars sense interactuar directament amb ells, sinó modificant l'entorn al seu voltant per afavorir certes accions o penalitzar-ne d'altres. Es podria moure endavant i endarrere en el temps per modificar, corregir i ajustar els elements que fan que en el futur l'imperi es col·lapsi. Per exemple, guiar-los a dipòsits de recursos útils o evitant que assentin les seves ciutats en llocs on en el futur hi hauran cataclismes.

Aquesta idea va ser descartada degut a la seva magnitud; faria falta un estudi complet per desenvolupar un videojoc d'aquestes característiques.

- Món obert:

Al jugador se'l presenta altra vegada amb un gran món pel que es pot moure. En aquest cas es tractaria d'una ciutat o poble habitada per planars. Cada una de les criatures seria un personatge únic amb les seves aspiracions, caràcter, defectes, etc, que interaccionarien entre ells i amb el seu entorn per dur a terme la seva vida dia a dia.

El jugador podria influir sobre els planars afectant el seu caràcter perquè reaccionin de maneres diferents a les situacions amb les que es troben. En aquest cas, el jugador no tindria un objectiu únic, sinó que la seva tasca consistiria en explorar els diversos resultats. Se'l podria presentar amb una llista de situacions que ha de aconseguir i se li donarien recompenses per cada una.

Altra vegada, aquesta idea va ser descartada per la seva ambició.

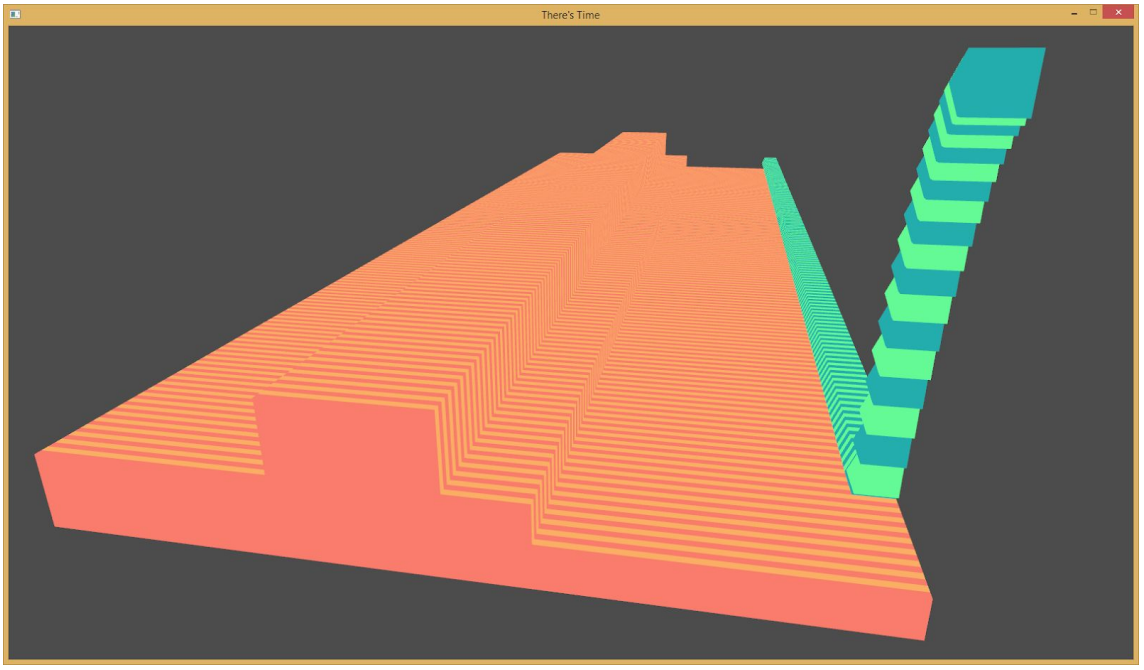
- Petits nivells:

La tercera idea consisteix en plantejar petits nivells amb un jugador i un planar. L'objectiu de cada nivell serà que el jugador guiï al planar fins a la meta. Inicialment, només el planar hi havia d'arribar però posteriorment es va veure que era una mecànica molt interessant el fet que el jugador també ho hagués de fer.

Es va triar aquesta idea per ser més manejable que les altres i perquè permetria provar el concepte d'una manera pura; sense soroll d'altres mecàniques com poden ser la gestió de recursos o la relació amb l'entorn. Es va optar per nivells auto-explicatius, que introduirien cada mecànica progressivament per reduir el numero de tutorials que fessin falta perquè el jugador entengués el nivell.

2.3- Primeres mecàniques

L'objectiu de la primera iteració era preparar un entorn sòlid tant pel jugador com pel planar i tenir a punt les interaccions més bàsiques. Al final d'aquest sprint, es va aconseguir el producte esperat:



Il·lustració 4: Captura de pantalla de la primera demo

El jugador flotava per l'espai i es podia moure a través dels sòlids, mentre que el planar estava restringit per la gravetat i col·lisions. Aquesta demo va servir per fer un pas endavant en el plantejament de mecàniques i va se la base sobre la que seguir desenvolupant.

Partint d'aquest producte es va fer un "brainstorm" de totes les mecàniques que es podríen aplicar a cada nivell. A continuació es presenta una llista de tot el que es va obtenir, tot i que no totes van acabar formant part del joc final:

Nom de la mecànica	**Efecte sobre el jugador**	**Efecte sobre el planar**
Col·lisions	No pot moure's a través de sòlids	No pot moure's a través de sòlids
Gravetat	Afectat	Afectat
Moviment	Pot moure's i saltar	Pot moure's i saltar
Enemics planars	No hi pot interactuar	Mor si entren en contacte
Pedres	Les pot moure per l'entorn 4D	Forma part dels elements amb els que col·lisiona

Zones de mort	Forma part dels elements amb els que col·lisiona	Mort
Distorsions	No pot interactuar amb el planar	Sense efecte
Plataformes	Forma part dels elements amb els que col·lisiona	Forma part dels elements amb els que col·lisiona
Caiguda	Sense efecte	Sense efecte
Interruptors	No pot interactuar	Els pot activar o desactivar
Temps	Sens efecte	Pot afegir temps al “final del nivell”
Checkpoints	Pot guardar el progrés a canvi de gastar temps del planar	Sense efecte
Salt de paret	Només pot saltar si es troba en una superfície horitzontal.	Pot saltar si toca una paret

Algunes d’aquestes mecàniques van ser descartades per diversos motius:

- Enemics planars: L’objectiu d’aquest joc es explotar la relació entre el jugador i el planar. El fet de posar criatures que matessin al planar però no afectessin al jugador no tenia sentit ja que per una banda contribuiria a convertir el joc en un *plattformer* comú i per l’altra, augmentaria la separació entre el jugador i el planar.

- Pedres: Aquesta mecànica va ser descartada per limitacions temporals. Es tractaria d’un element que el jugador podria moure a llocs convenients perquè el planar pogués completar el nivell.

- Interruptors: Altra vegada limitacions temporals. Construir el sistema d’interruptors que iniciessin o aturessin el moviment de plataformes o altres elements hauria implicat una gran càrrega de treball ja que s’haurien hagut de fer canvis a totes les parts del joc; des de l’estructura de nivells fins a les físiques.

- Temps: Aquesta mecànica consistia en què en el nivell es trobarien uns objectes que en ser tocats pel planar afegirien més instants al final del nivell, farien que fos més llarg. Es va descartar per complicacions a l’hora de fer-la encaixar amb els altres elements.

- Checkpoints: Es va descartar per la mida dels nivells i complicacions amb les interaccions amb altres mecàniques. La idea inicial era que

servís al jugador per no perdre tot el progrés si queia al buit a canvi "gastar" temps del planar, és a dir forçar que estigués quiet durant un curt període de temps.

2.4- Desenvolupament de mecàniques

Nivell sòlid i restricció del jugador a l'entorn físic

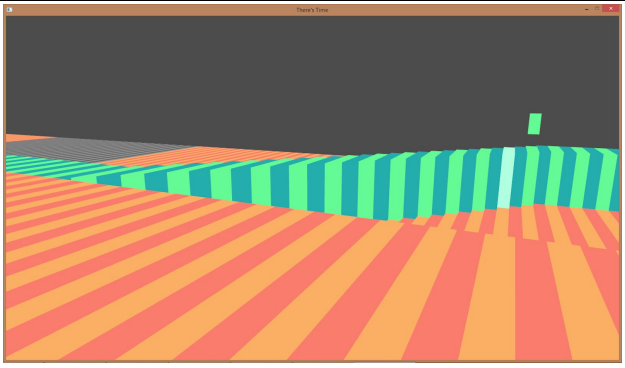
Inicialment, el jugador havia de ser capaç de volar i travessar els elements sòlids del nivell, guiant al planar fins al final. Després de la primera demo, es va veure clarament que la mecànica clau del joc seria la relació entre el jugador i el planar a través de la diferent interacció amb els mateixos elements.

Interaccions entre el jugador i el planar

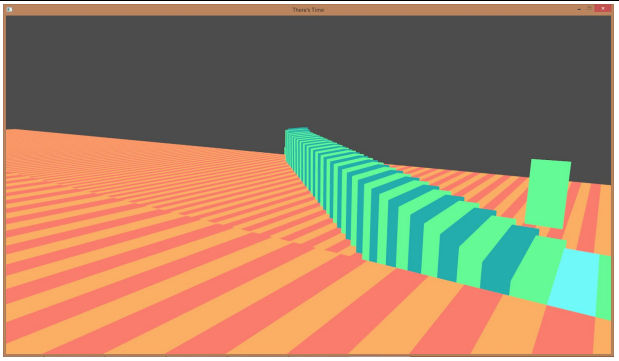
Cada joc ha de tenir una idea bàsica al voltant de la qual és construït. En aquest cas, inicialment semblava que la mecànica clau fos el fet que el temps del planar és espai pel jugador però després dels primers testos va quedar clar que un joc així estaria faltat d'un nivell de complexitat que el fes interessant.

Accions del planar

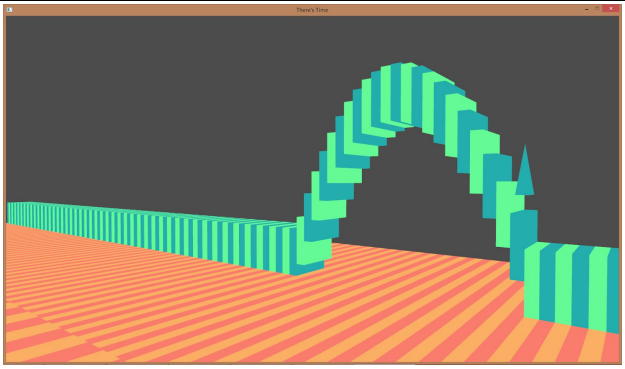
El planar podrà rebre ordres del jugador. Aquestes ordres seran: avançar, retrocedir, saltar i cancel·lar el moviment o el salt.



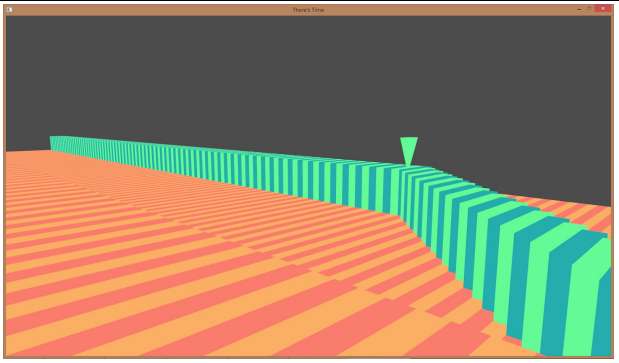
Il·lustració 5: El planar avança



Il·lustració 6: El planar retrocedeix



Il·lustració 7: El planar salta

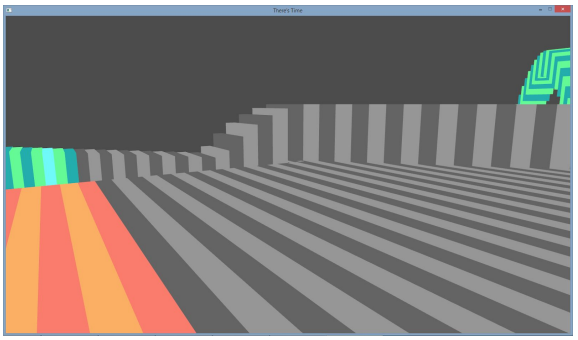


Il·lustració 8: El planar atura el seu moviment

Per facilitar la lectura per part de l'usuari, cada ordre que rebi el planar serà indicada amb una fletxa que assenyalarà cap a quina direcció es fa el moviment.

Distorsions

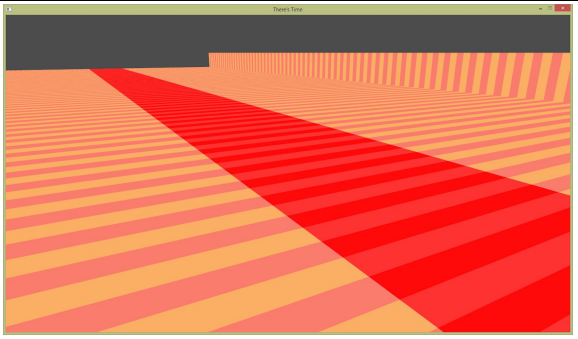
El primer element que afecta la relació entre el jugador i el planar. Son instants en els que el jugador no podrà donar ordres al planar, que seguirà sotmès als efectes de la gravetat, col·lisions, zones de mort, etc.



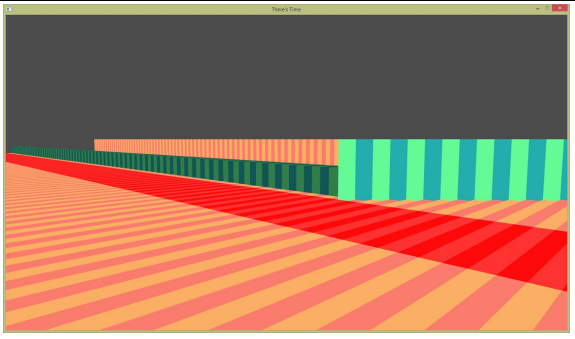
Il·lustració 9: Distorsions

Zones de mort

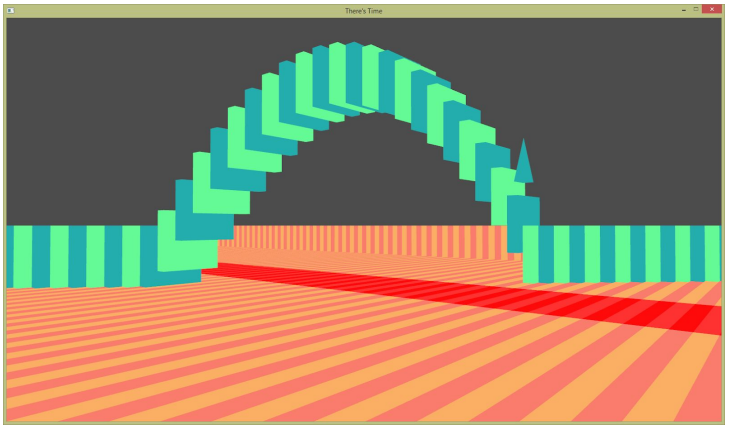
Es tracta de superfícies per les quals el jugador pot caminar sense problemes però que mataran al planar. Es representen amb el color vermell.



Il·lustració 10: Zones de mort, representades en vermell



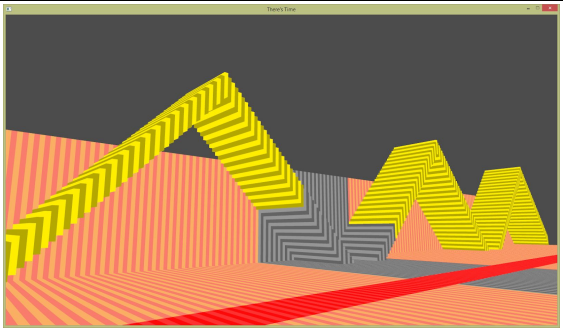
Il·lustració 11: El planar mor en tocar-les



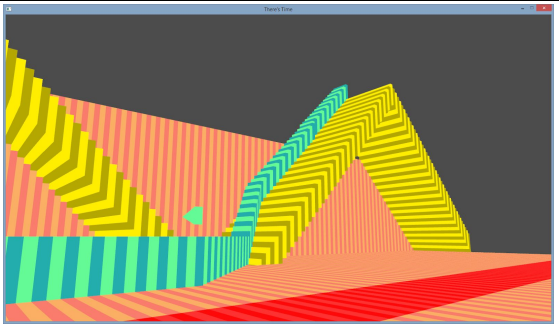
Il·lustració 12: El planar pot evitar aquestes zones si salta a temps

Plataformes

Es tracta d'un altre dels elements diferenciadors en la relació entre el agents. Pel planar, les plataformes son sòlids que es mouen de manera cíclica, mentre que pel jugador son superfícies per les que pot avançar:



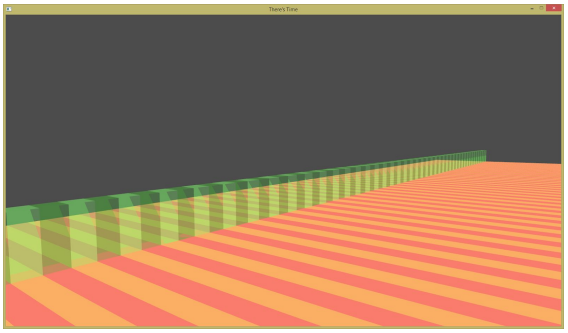
Il·lustració 13: Plataformes afectades per distorsions



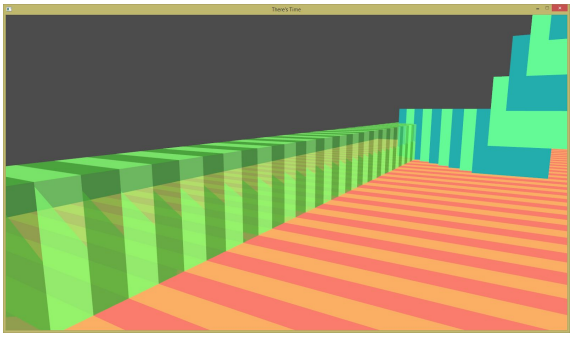
Il·lustració 14: El planar puja per una plataforma

Meta

Cada nivell tindrà la seva meta, representada per una figura semitransparent de color verd. Quan un dels dos agents la toqui, canviarà de forma i transparència per donar el missatge que s'ha realitzat progrés.



Il·lustració 15: Meta

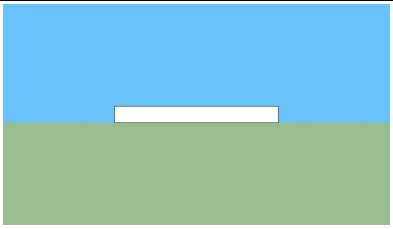


Il·lustració 16: Canvia de propietats en ser tocada

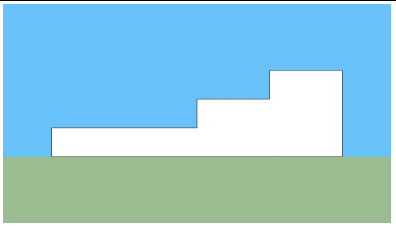
2.5- Introducció de mecàniques

Les mecàniques s'introduiran progressivament per ordre de complexitat. Començant pel moviment bàsic i a continuació les plataformes, les zones de mort i les distorsions. Es presentarà cada mecànica per separat en el seu propi nivell i s'anirà afegint complicacions. Quan una s'hagin esgotat les combinacions s'introduirà el següent concepte.

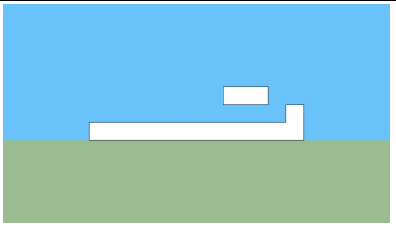
El moviment del jugador i el planar s'explica en els primers 7 nivells, que van augmentant la seva complexitat progressivament:



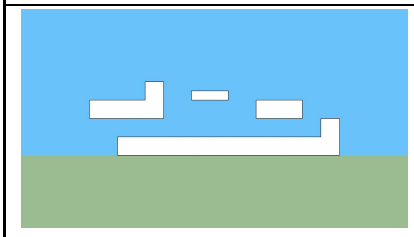
Il·lustració 17: Nivell 1



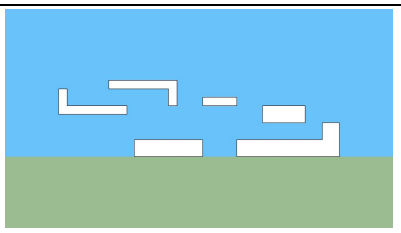
Il·lustració 18: Nivell 2



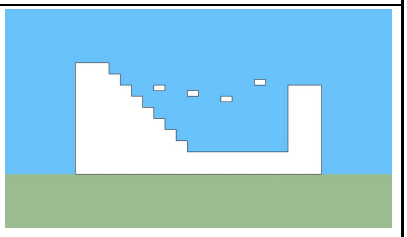
Il·lustració 19: Nivell 3



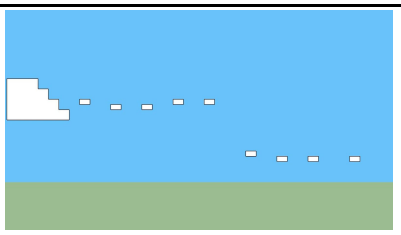
Il·lustració 20: Nivell 4



Il·lustració 21: Nivell 5

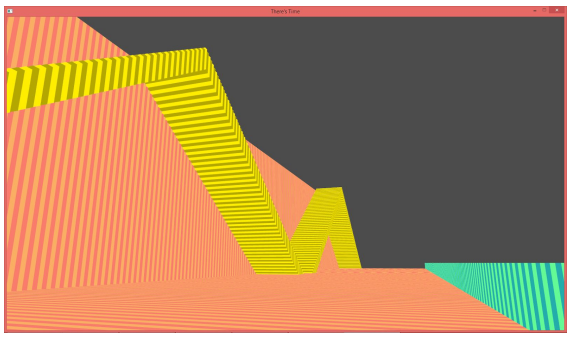


Il·lustració 22: Nivell 6

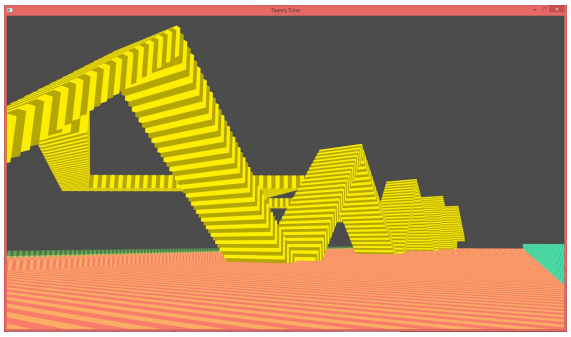


Il·lustració 23: Nivell 7

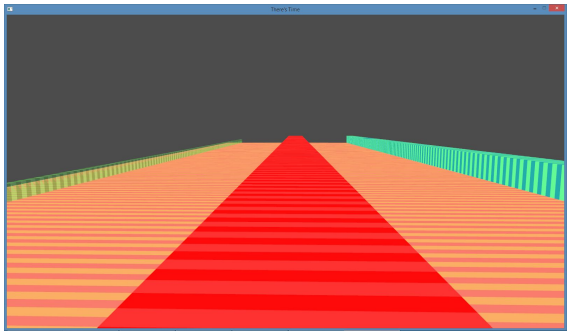
Inicialment els nivells no tenen cap risc, el jugador pot experimentar amb els moviments del planar i els seus propis. A partir del nivell 5 entra el primer risc, es tracta d'un forat que s'haurà de saltar. A continuació es demanen un salts de precisió; altra vegada sense risc i en el següent nivell el jugador ha repetir els salts però aquesta vegada pot caure al buit.



La primera mecànica introduir seran les plataformes. En aquest nivell, l'únic objectiu serà fer pujar el planar per la plataforma i seguir-lo.

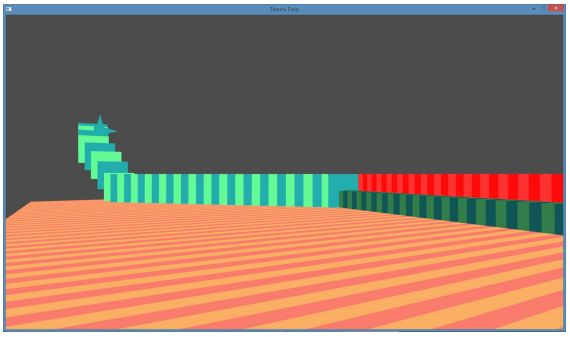


El següent nivell afegirà un grau de complicació, fent que s'hagi de saltar d'una plataforma a l'altra

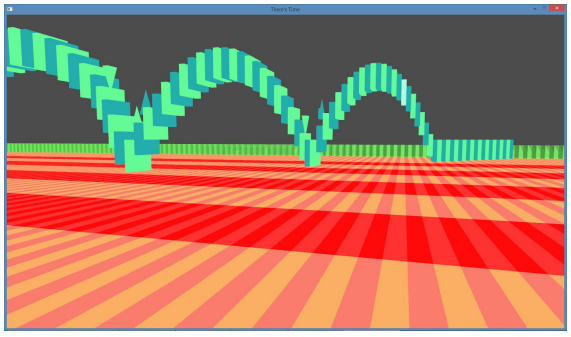


Per introduir la següent mecànica, les zones de mort, hem de mostrar a al jugador que a ell no li afecten. Perquè ho vegi, farem que aparegui directament sobre d'elles:

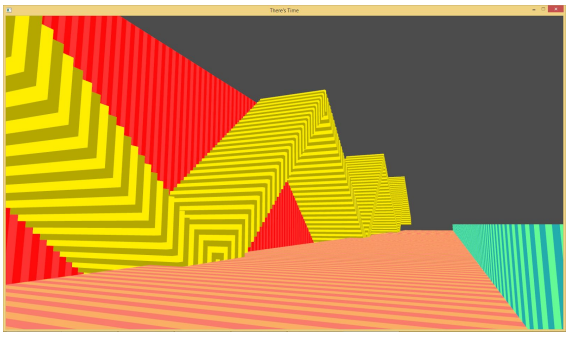
Després, ha de veure que si que afecta al planar. Aixo ho aconseguirem fent que el planar xoqui contra aquesta zona amb la primera ordre que se li doni:



D'aquesta manera el jugador veurà el que passa i podrà ajustar el moviment per fer que salti per sobre.

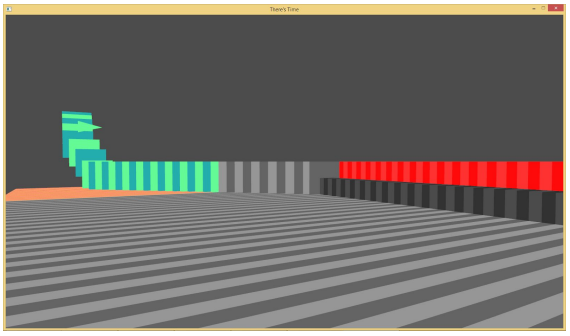


En el següent nivell, aprofundirem en la idea que el planar ha d'esquivar aquestes zones. Podem veure que en aquesta part, el jugador pot passar sense cap problema mentre que el planar ha de saltar.

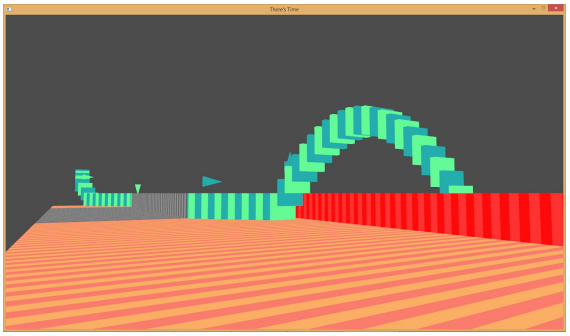


En el següent nivell, combinarem les plataformes i les zones de mort:

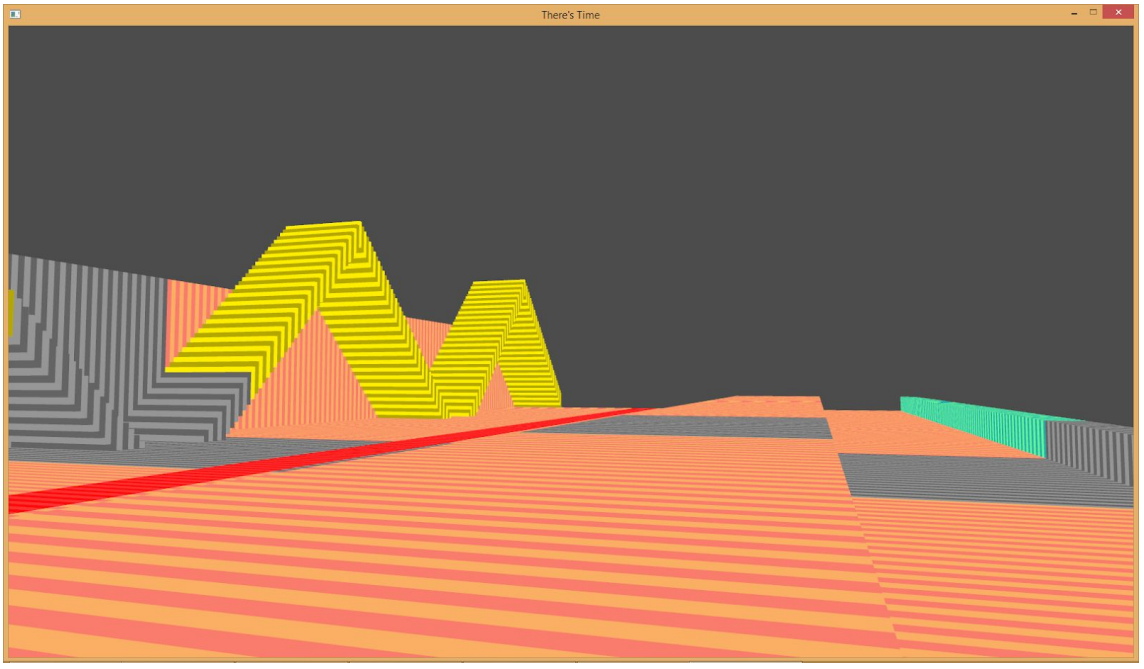
El planar haurà de saltar de plataforma en plataforma esquivant aquestes arees.



Les distorsions les introduïrem de la mateixa manera que les zones de mort. Aquesta vegada, el jugador veurà que el nivell li és familiar i repetirà la mateixa estratègia, només que aquesta vegada s'adonarà que no pot fer saltar el planar en el lloc que li convé. Li haurà de donar la ordre de quedar-se quiet i fer-lo saltar després de la distorsió.



En els nivells posteriors es combinaran aquestes mecàniques.



2.6- Aspectes tècnics

Estructura del codi

El codi s'estructura de manera modular, és a dir, cada part del codi s'encarrega únicament d'una tasca. Això permet escalar el projecte més fàcilment i simplifica en gran mesura el desenvolupament de noves funcionalitats i la resolució de bugs i problemes.

El joc està dividit en 4 mòduls:

- Renderitzador
- Escena
- Fisiques
- Nivell

Addicionalment tenim la classe principal que s'encarrega d'iniciar el joc i gestionar les interaccions entre els mòduls. Cal destacar que aquests mòduls estan formats per diverses clases:

- Renderitzador:
  - Shader, classe C++
  - Vertex i fragment shader, arxius GLSL
- Escena
  - Nivell, classe C++
  - Jugador, classe C++
  - Planar, classe C++
- Nivell
  - Plataformes, classe C++
  - Superfícies mortals, classe C++

La classe destinada a les físiques no utilitza cap altre classe addicional.

Funcionament general i renderitzat

La classe main.cpp s'encarrega d'iniciar el programa instanciant objectes per l'escena i el renderitzador, així com els callbacks pels diversos inputs que pot fer l'usuari:

```
void processKeyInput(GLFWwindow *window, int key, int scancode, int action, int mode);
void processMouseInput(GLFWwindow* window, double xpos, double ypos);
void processScrollInput(GLFWwindow* window, double xoffset, double yoffset);

void loadLevel(std::string level_name);

Renderer * renderer_obj;
SceneManager * scene_manager;

float deltaTime = 0.0f;
float lastFrame = 0.0f;
```

```
float lastX, lastY;
bool firstMouse;
```

A continuació, es defineix la llista de nivells, els diferents tipus de renderitzat i el mode de render actual. Aquests modes són:

- WELCOME_SCREEN: Sobreposarà un text de benvinguda al jugador abans de començar a jugar el primer nivell, el jugador podrà prémer qualsevol botó per continuar.
- LEVEL_INIT: Abans de començar el nivell es mostrarà un text indicant quin és el seu nom o número.
- REGULAR_LEVEL: Indicarà que l'usuari està jugant el nivell i mostrarà els textos que pertanyin al nivell.
- LEVEL_COMPLETE: Un cop el jugador hagi completat el nivell es mostrarà el missatge pertinent i se li demanarà que toqui qualsevol tecla per continuar al següent nivell
- ENDGAME: Un cop el jugador hagi arribat a l'últim nivell se li mostrarà un text indicant que ha acabat el joc i que pot prémer una tecla per tornar a començar o ESC per sortir.

```
std::vector<std::string> level_list = {
	"level1.lvl",
	"level2.lvl",
	"level3.lvl",
	"level4.lvl",
	"level5.lvl",
	"level6.lvl",
	"level7.lvl",
	"level8.lvl",
	"level9.lvl",
	"level10.lvl"
};

enum RENDER_SET {
	WELCOME_SCREEN,
	LEVEL_INIT,
	REGULAR_LEVEL,
	LEVEL_COMPLETE,
	ENDGAME
};

RENDER_SET active_rendering = WELCOME_SCREEN;

int current_level = 0;
```

A continuació s'inicialitzaràn diverses variables necessàries perquè OpenGL pugui crear una finestra pel joc. Forçarem que s'utilitzi la versió 3 per evitar que hi puguin haver errors de versions, evitarem que la finestra pugui ser ajustable i activarem antialiasing. Aquesta finestra s'assignarà al renderitzador com la finestra per la que han de sortir les imatges; s'indicarà la seva mida i

títol. Seguidament, es demanarà al renderitzador que prepari les altres configuracions necessàries per poder mostrar els frames correctament.

```
glewExperimental = GL_TRUE;
glfwInit();
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);

glEnable(GL_MULTISAMPLE);
glfwWindowHint(GLFW_SAMPLES, 4);

GLFWwindow* window = glfwCreateWindow(Renderer::SCR_WIDTH, Renderer::SCR_HEIGHT, "There's Time",
nullptr, nullptr); // Windowed;

glfwMakeContextCurrent(window);

glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

glewInit();

glfwSetKeyCallback(window, processKeyInput);
glfwSetCursorPosCallback(window, processMouseInput);

glViewport(0, 0, Renderer::SCR_WIDTH, Renderer::SCR_HEIGHT);

renderer_obj = new Renderer();
renderer_obj->window = window;
renderer_obj->setUpTextRendering();
```

En inicialitzar en Renderitzador es duen a terme diversos processos. Primerament es carreguen els shaders i si no hi ha cap error es retorna el codi 0:

```
bool Renderer::setUpShaders() {
   this->active_shader.buildShader(this->vertex_shader, this->fragment_shader);
   this->active_text_shader.buildShader(this->text_vertex_shader, this->text_fragment_shader);
   return 0;
}
```

Aquest procés es troba a la classe Shader i el que fa és llegir l'arxiu que se li indica, que en aquest cas son shaders/default_shader.vs i shaders/default_shader.frag per les imatges i shaders/text_default_shader.vs i shaders/text_default_shader.frag pels textos. Per simplicitat d'ús i de codi, es faran servir dos jocs de shaders, un per renderitzar l'escena i l'altre únicament pels textos, tot i que seria possible fer-ho en un de sol.

Cal tenir en compte que l'ús de múltiples shaders pot augmentar el temps total que s'està renderitzant però com que el projecte és petit i els programes GLSL no tenen més d'una vintena de línies, canviar de shader és un procés molt ràpid.

El procés de càrrega i compilació és lent i es fa només una vegada quan s'incia el joc. Un cop el programa està compilat s'assigna a un slot per programes de OpenGL:

```
const GLchar* vShaderCode = vertexCode.c_str();
const GLchar * fShaderCode = fragmentCode.c_str();
// 2. Compile shaders
GLuint vertex, fragment;
// Vertex Shader
vertex = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex, 1, &vShaderCode, NULL);
glCompileShader(vertex);
checkCompileErrors(vertex, "VERTEX");
// Fragment Shader
fragment = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragment, 1, &fShaderCode, NULL);
glCompileShader(fragment);
checkCompileErrors(fragment, "FRAGMENT");
// If geometry shader is given, compile geometry shader
GLuint geometry;
if (geometryPath != nullptr)
{
     const GLchar * gShaderCode = geometryCode.c_str();
     geometry = glCreateShader(GL_GEOMETRY_SHADER);
     glShaderSource(geometry, 1, &gShaderCode, NULL);
     glCompileShader(geometry);
     checkCompileErrors(geometry, "GEOMETRY");
}
// Shader Program
this->Program = glCreateProgram();
glAttachShader(this->Program, vertex);
glAttachShader(this->Program, fragment);
if (geometryPath != nullptr)
    glAttachShader(this->Program, geometry);
glLinkProgram(this->Program);
checkCompileErrors(this->Program, "PROGRAM");
// Delete the shaders as they're linked into our program now and no longer necessery
glDeleteShader(vertex);
glDeleteShader(fragment);
if (geometryPath != nullptr)
     glDeleteShader(geometry);
```

El shader de geometria es un programa addicional que permet post-procesar la imatge generada, en el cas que ens implica no el necessitem.

Seguidament, inicialitzarem l'escena i li indicarem que carregui el primer nivell; es carregarà el model i els diversos elements i el gestor d'escena s'encarregarà de crear múltiples instants, un per cada diferencial de temps que tingui el nivell. (Aquest pas s'explicarà en detall en el següent punt). Comptarem el temps que ha passat des de l'últim frame per poder calcular les

físiques a temps real al jugador i segons el tipus de renderitzant que s'estigui fent es fixarà la posició del jugador o es demanarà al gestor d'escena que actualitzi les físiques.

```
scene_manager = new SceneManager();

loadLevel(level_list.at(current_level));

// Game loop
while (!glfwWindowShouldClose(renderer_obj->window))
{
    GLfloat currentFrame = glfwGetTime();
    deltaTime = currentFrame - lastFrame;
    lastFrame = currentFrame;

    scene_manager->delta_time = deltaTime;

    if (active_rendering == WELCOME_SCREEN) {
                scene_manager->active_player.Position.x = scene_manager->current_level.player_starting_position.x +
sin(currentFrame / 50);
    }

    if (active_rendering == REGULAR_LEVEL) {
         scene_manager->ProcessPlayerPhysics();
         scene_manager->UpdateInstant();
    }

       // Check and call events
    glfwPollEvents();

    renderer_obj->renderScene(scene_manager, active_rendering);

    if (scene_manager->active_player.Position.y < scene_manager->current_level.player_fall_reset_threshold) {
                scene_manager->active_player.Position =
glm::vec3(scene_manager->current_level.player_starting_position.x,
scene_manager->current_level.player_starting_position.y, 3);

         scene_manager->active_player.Pitch = 0;
         scene_manager->active_player.Yaw = 0;

          scene_manager->active_player.updateCameraVectors();
    }

    /*
     * Condició per completar el nivell
     */
    if (scene_manager->player_completed && scene_manager->flatter_completed) {
         active_rendering = LEVEL_COMPLETE;
         scene_manager->active_player.Yaw = 90;
         scene_manager->active_player.Pitch = 0;
         scene_manager->active_player.updateCameraVectors();
    }
}
```

Si el jugador ha caigut per sota del límit del nivell, se'l teletransportarà altre cop a l'inici i si la condició per completar el nivell s'ha assolit, s'actualitzarà el format de renderitzat per mostrar-ho al jugador.

Durant aquest bucle es va cridant el renderitzador amb l'escena que s'ha de dibuixar i el format amb el que s'ha de fer. S'accedirà a l'escena a través d'un punter de memòria, ja que conté tots els instants del nivell i es tracta d'un objecte prou gran:

```
int Renderer::renderScene(SceneManager * scene, int render_mode)
{

	glClearColor(0.3f, 0.3f, 0.3f, 1.0f);
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	glEnable(GL_DEPTH_TEST);

	glViewport(0, 0, this->SCR_WIDTH, this->SCR_HEIGHT);

	this->active_shader.Use();

	glm::mat4 projection = glm::perspective(glm::radians(scene->active_player.Zoom),
(float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
	glm::mat4 view = scene->active_player.GetViewMatrix();

	this->active_shader.setMat4("projection", projection);
	this->active_shader.setMat4("view", view);

	int vertexColorLocation = glGetUniformLocation(this->active_shader.Program, "ourColor");
	glUseProgram(this->active_shader.Program);
```

Començarem netejant la imatge que haviem renderitzat en el cicle anterior indicant `glClearColor(0.3f, 0.3f, 0.3f, 1.0f);` També ens servirà per pintar el fons de l'escena. Immediatament després buidarem els buffers de color i profunditat per no arrossegar artefactes del cicle anterior i activarem la comprovació de profunditat perquè els shaders dibuixin els objectes correctament.

Sense GL_DEPTH_TEST els shaders dibuixarien cada objecte en el moment en que els arriba, sense tenir en compte quin objecte es davant de quin o quina cara de l'objecte es la que el jugador ha de veure. Amb aquesta variable, és dur a terme un pas extra que tindrà en compte les questions que acabem de mencionar.

A continuació s'assignen els colors a cada un dels elements del joc, com hem comentat abans, els colors son:

- Nivell -> Carbassa
- Planar -> Turquesa
- Plataformes -> Groc
- Zones de mort -> Vermell
- Distorsions -> Gris
- Meta -> Verd

Cada un d'aquests colors s'assigna amb una variació per mostrar clarament una distinció entre els instants. Per cada un d'aquests instants es renderitzarà per ordre:

- El sòlid del nivell
- El planar
- La fletxa indicativa de les accions del planar
- Les plataformes
- Les zones de mort
- La meta
- Els textos que pertoquin

La meta s'ha de dibuixar en l'últim moment ja que conté transparència i si es dibuixés al principi crearia un efecte incorrecte, ja que a través d'aquest objecte no es veuria el que hi ha a l'altra banda, sinó el fons de l'escena. Per últim, s'intercanvien els buffers.

```
glfwSwapBuffers(this->window);
```

OpenGL fa servir un sistema de buffer doble o triple, en el nostre cas triarem el tipus doble. Aquest sistema té dos buffers, tapíssos sobre els que pot dibuixar, mentres es dibuixa sobre un, es mostra l'altre per pantalla. Si no fos així el jugador veuria per pantalla un "Flickering", alguns objectes farien pampallugues o estiarien aparentment incomplets degut a que el renderitzador estava a mig dibuixar quan la imatge s'ha emès per pantalla.

Estructura i càrrega de nivells

Cada nivell està compost per múltiples elements que es carreguen a través d'un arxiu de text pla amb extensó lvl. Aquests arxius tenen la següent forma:

```
player_position, 7,4
flatter_position,2,1.5
goal_position,19,4
level_length,300
distortion_list, 75, 120
level_obj,lvl1.obj
player_fall_reset_threshold,-50
platform,platform_1.obj,15,0,15,5,0.02
death_surface,lvl1_ds.obj
text,8.5,2,100,Heu d'arribar a la meta
```

Cada una d'aquestes línies descriu un element del nivell separant el nom de l'element i els diversos camps necessaris per comes:

- `player_position:` Emmagatzema la posició incial del jugador en el nivell, en coordenades x i y, també és la posició en la que apareixeria si caigués al buit.

- `flatter_position:` Emmagatzema la posició incial del planar, en coordenades x i y, a partir de la qual començarà a moure's pel nivell.

- `goal_position:` Emmagatzema la posició de la meta, en coordenades x i y.

- `level_length:` Indica el número d'instants que ha de tenir el nivell. Com més gran sigui aquest número més temps tindrà el planar per arribar a la meta.

- `distortion_list:` Emmagatzema l'instant inicial i final que han de contenir distorsions. En aquest cas, significa que entre els instants 75 i 120 el nivell tindrà una distorsió. En l'arxiu hi poden haver múltiples lines d'aquest tipus per un nivell que tingui més d'una distorsió.

- `level_obj:` Indica l'arxiu de tipus obj que contindrà el solid base del nivell.

- `player_fall_reset_threshold:` Llindar a partir del cual, si el jugador hi cau, es reiniciarà el nivell. És important que sigui un valor personalitzable ja que és possible que algun nivell tingui la meta per sota del valor que hi podria haver per defecte, de manera que no es podria completar.

- `platform:` Indica una de les múltiples plataformes que pot tenir el nivell. Els camps que requereix son, per ordre, l'arxiu en format obj que conté el sòlid de la plataforma, la posició inicial en el cicle de moviment de la plataforma, la posició final en el cicle de moviment i la velocitat amb la que ja de fer aquest moviment. Aquests elements no tenen cap limitació, una plataforma podria estar composta per un sòlid de la mateixa mida que un nivell i moure's paralela als eixos o en diagonal.

- `death_surface:` Indica l'arxiu de tipus obj que contindrà el sòlid que formen les superfícies mortals. Ha d'estar escalat i encaixar amb el l'obj del nivell base.

- `text:` Indica el llistat de textos que ha de mostrar el nivell. Per ordre, els seus camps son: coordenades x i y de la seva posició, número de l'instant en el que s'ha de mostrar i contingut del text.

La simplicitat i escalabilitat d'aquest format permet crear els nivells de manera molt senzilla, utilitzant qualsevol programa que pugui exportar en format obj com pot ser SketchUp, Autocad, Blender, 3dsMax, etc. Aquesta simplicitat es la que obre les portes a la comunitat, es podria afegir un mode de

joc que permetés jugar als nivells creats per altres usuaris, cosa que allargaria enormement la vida i possibilitats del joc.

El procés de càrrega es comença en el main.cpp ja que hi interven múltiples mòduls.

```
void loadLevel(std::string level_name) {

    scene_manager->LoadLevel(level_list.at(current_level));

    scene_manager->active_player =
Player(glm::vec3(scene_manager->current_level.player_starting_position.x,
scene_manager->current_level.player_starting_position.y, 2), glm::vec3(0, 1, 0), 90, 10);
    scene_manager->active_player.displacement =
glm::vec3(scene_manager->current_level.flatter_center.x,
scene_manager->current_level.flatter_center.y * 1.5f, -0.1f);

    renderer_obj->updateLevelModel(&scene_manager->current_level);

    scene_manager->UpdateTimeline();
}
```

Per començar, el gestor d'escena elimina el nivell que tingui en memòria, si es que n'hi ha cap i el substitueix pel nou. A continuació s'esborren les accions que el jugador hagi indicat en el nivell anterior i s'eliminen els instants.

```
void SceneManager::LoadLevel(std::string level_name)
{
	this->current_level = * new Level(level_name);
	this->timeline_length = this->current_level.level_length;
	for (int i = 0; i < this->current_level.level_length; i++){
		this->order_list.push_back(ORDER_LIST::ORDER_NONE);
		this->action_list.push_back(ACTION_LIST::ACTION_NONE);
	}

	this->ResetActions();
	this->timeline.clear();
}
```

El procés de càrrega del propi nivell consisteix en anar iterant l'arxiu que el defineix, l'ordre de les línies es irrellevant, i anar assignant les variables en el camp que pertoqui. Per carregar un arxiu obj es crida una funció auxiliar que s'encarrega de llegir l'arxiu i construir els triangles formats pels vèrtexs i assignar els que pertoquin al llistat de sòlids amb el que es pot colisionar.

Un cop el nivell ha estat llegit i creat, main.cpp li envia al Renderitzador perquè carregui el model en memòria.

```
GLuint modelVAO, modelVBO;

glGenVertexArrays(1, &modelVAO);
glGenBuffers(1, &modelVBO);
// Fill buffer
glBindBuffer(GL_ARRAY_BUFFER, modelVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(current_level->modelVertices), current_level->modelVertices,
GL_STATIC_DRAW);
// Link vertex attributes
glBindVertexArray(modelVAO);
glEnableVertexAttribArray(0);
//(vertex atribute array to store, number of values, isNormalized?, size of vertex data, offset
from start of vertex data)
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)(3 *
sizeof(GLfloat)));
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)(6 *
sizeof(GLfloat)));
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);

current_level->VAO = modelVAO;
current_level->VBO = modelVBO;
```

Aquest procés s'encarrega generar un id de VAO i VBO; Vertex Attribute Object i Vertex Buffer Object. Es vincula l'id de VAO i VBO i es generen la llista d'atributs i els buffers respectivament. A continuació es vincula el buffer i se li carreguen les dades extretes anteriorment de l'arxiu obj indicant les pròpies dades i la seva mida en bits.

Aquestes dades es vinculen als diversos arrays d'atributs del VAO indicant els punts en l'espai, coordenades de textures i vectors normals, tot i que en aquest cas, nosaltres no farem servir ni textures ni normals.

Un cop el model està carregat, se li assignen els id's de VAO i VBO per poder accedir-hi durant el procés de renderitzat.

Detecció de col·lisions i físiques

En cada una de les iteracions s'han de calcular les físiques que afecten al jugador i en el cas que donés una ordre al planar, també s'haurien d'actualitzar els instants adequats. El sistema que s'encarrega de trobar les col·lisions es el mateix en els dos casos, cosa que ha reduït en gran mesura la càrrega del desenvolupament.

Tot i amb això, el principi del procés és una mica diferent. Quan el jugador dóna una ordre al planar s'indica en la llista d'ordres i es guarda en quin instant s'ha donat, per poder minimitzar la quantitat de càlculs que s'han de fer. Durant el cicle normal del nivell es crida cada vegada aquestes dues funcions:

```
if (active_rendering == REGULAR_LEVEL) {
     scene_manager->ProcessPlayerPhysics();
     scene_manager->UpdateInstant();
}
```

S'encarreguen de calcular les físiques del jugador i d'actualitzar els instants que siguin necessaris. Comencem per les físiques del planar.

Quan es carrega un nivell, el gestor d'escena guarda l'original i crea tantes còpies del nivell com instants té. Cada copia es fa de l'instant anterior, no de l'instant original per mantenir una coherencia temporal. Com hem comentat abans, quan el jugador dóna una ordre, aquesta s'assigna a l'instant que li correspon i la posició d'aquest l'instant s'emmagatzema. A la classe main, a l'hora d'actualitzar els instants es comprova el valor d'aquesta variable i s'actualitza l'instant que indica. Un cop s'ha acabat el procés es suma 1 a aquesta variable perquè en la següent iteració s'actualitzi el següent instant. El número també es fa servir a l'hora de renderitzar per no mostrar el planar més endavant de l'instant que s'està actualitzant.

Aquest és el codi que s'encarrega de dur a terme l'actualització. Comença copiant les dades necessàries de l'instant anterior, com poden ser la posició o el moment d'inèrcia del planar i es comprova que l'instant no formi part d'una distorsió.

```
if (instant_index > 0 && instant_index < this->current_level.level_length - 1) {
	Level * last_level = &this->timeline.at(instant_to_update - 1);
	Level * current_instant = &this->timeline.at(instant_to_update);

	current_instant->flatter_position = last_level->flatter_position;
	current_instant->flatter_momentum = last_level->flatter_momentum;

	current_instant->platform_list = last_level->platform_list;
	current_instant->deathSurface_list = last_level->deathSurface_list;

	bool distort = false;
	for (int c = 0; c < current_instant->distortion_init_list.size(); c++)
	{
		if (instant_index > current_instant->distortion_init_list.at(c) && instant_index <
current_instant->distortion_end_list.at(c)) {
			distort = true;
		}
	}
```

A continuació es calcula quina és l'acció que ha de fer el planar. Si l'instant actual no en té cap però anteriorment se li havia dit que havia

d'avançar o retrocedir, es mantindrà aquella ordre. Un cop s'han assignat les variables necessàries, s'envia l'instant en el seu estat incial al mòdul de físiques. Aquest rebrà l'estat incicial i en calcularà l'estat final. Modificarà l'objecte que rebi en comptes de retornar-ne un de nou per evitar sobrecàrregues de memòria.

```
ORDER_LIST order_definitive;

if (distort) {
	current_instant->action = ACTION_LIST::ACTION_NONE;
}
else {
	current_instant->action = this->action_list.at(instant_index);;
}

if (this->order_list.at(instant_index) == ORDER_LIST::ORDER_NONE || distort) {
	order_definitive = last_level->order;
}
else {
	order_definitive = this->order_list.at(instant_index);
}
current_instant->order = order_definitive;

Physics::ProcessInstant(current_instant);
```

Al seu torn, el primer que es fa al mòdul de físiques és calcular el moviment que ha de fer el planar a partir del moment que portava acumulat i l'acció que se li indiqui amb una suma i l'equació d'un moviment uniforme. Aquesta acció es realitza tenint en compte si el planar és viu o mort i si està tocant una superficie solida o no.

A continuació es calcula la posició final del planar amb l'equació del moviment uniformement accelerat i s'hi apliquen els modificadors de l'ordre. Després, s'envia a la funció encarregada de detectar les col·lisions:

```
glm::vec2 initial_position = instant->flatter_position;
//Pf = Pi + Vi*t + 1/2*a*t^2;
glm::vec2 end_position = initial_position +
		end_momentum * instant->instant_length +
		instant->gravity * pow(instant->instant_length, 2) / 2.0f;

if (instant->order == ORDER_LIST::FORWARDS) {
	end_position.x += instant->flatter_speed;
}

if (instant->order == ORDER_LIST::BACKWARDS) {
	end_position.x -= instant->flatter_speed;
}

Physics::calculate2dColision(initial_position, end_position, end_momentum, instant);
```

Aquesta funció rep la posició inicial del planar, la posició final, el moment final i l'instant inicial, que contindrà la llista de sòlids amb els que colisionar i altres dades d'importància. Les colisions es calcularàn en dues parts; la primera serà amb tots els objectes sòlids del nivell, incloent les plataformes en el seu estat inicial. Després s'aplicarà el moviment de les plataformes i s'ajustarà la posició final del planar en conseqüència.

```
std::vector<glm::vec2> outer_hull = current_level->flatter_flatterOuterHull;
std::vector<Triangle> solid_list = current_level->solid_list;

//Afegim les zones de mort als solids amb els que es pot colisionar.
for (std::vector<DeathSurface>::iterator ds = current_level->deathSurface_list.begin(); ds != 
current_level->deathSurface_list.end(); ds++) {
       solid_list.insert(solid_list.begin(), ds->solid_list.begin(), ds->solid_list.end());
}

glm::vec2 move_vec = final_position - initial_position,
       flatter_center = current_level->flatter_center + initial_position;

glm::vec2 init_point, end_point;
current_level->debug_render_text.push_back(std::make_pair(current_level->flatter_position + 
glm::vec2(0, 1.0f), std::to_string(initial_position.x) + ", " + 
std::to_string(initial_position.y)));

for (std::vector<Platform>::iterator pl = current_level->platform_list.begin(); pl != 
current_level->platform_list.end(); pl++) {

   for each (Triangle tr in pl->solid_list)
   {
         Triangle t_prima = Triangle(
             tr.point_1 + pl->current_position,
             tr.point_2 + pl->current_position,
             tr.point_3 + pl->current_position
          );
          solid_list.push_back(t_prima);
     }
}
```

Aquest tros de codi ens dona la variable solid_list, que conté un llistat de triangles en la seva posició absoluta. Seran tots els objectes amb els que es podrà colisionar.

Les colisions es calcularàn en dues fases. La primera serà comprovar que existeixi el xoc i la segona serà calcular el punt final tenint-lo en compte.

Prèviament, durant la càrrega del nivell haurem preparat un model pel planar. Un dels passos que s'haurà dut a terme és el de trobar la closca convexa del model, és a dir, els punts més externs. Ens servirà per saber quins son els límits de la criatura i farem servir aquests punts com a referència per fer els càlculs.

En el codi següent iterarem cada un dels punts de la closca convexa i revisarem que si els movem segons el vector de moviment que hem calculat abans no colisioni amb cap dels triangles que formen part de la llista d'elements sòlids. La primera funció, `doIntersect`, s'encarrega de comprovar si el punt en questió xoca i si ho fa es crida la segona funció, `processIntersection`, que ens retornarà el punt on els objectes colisionen.

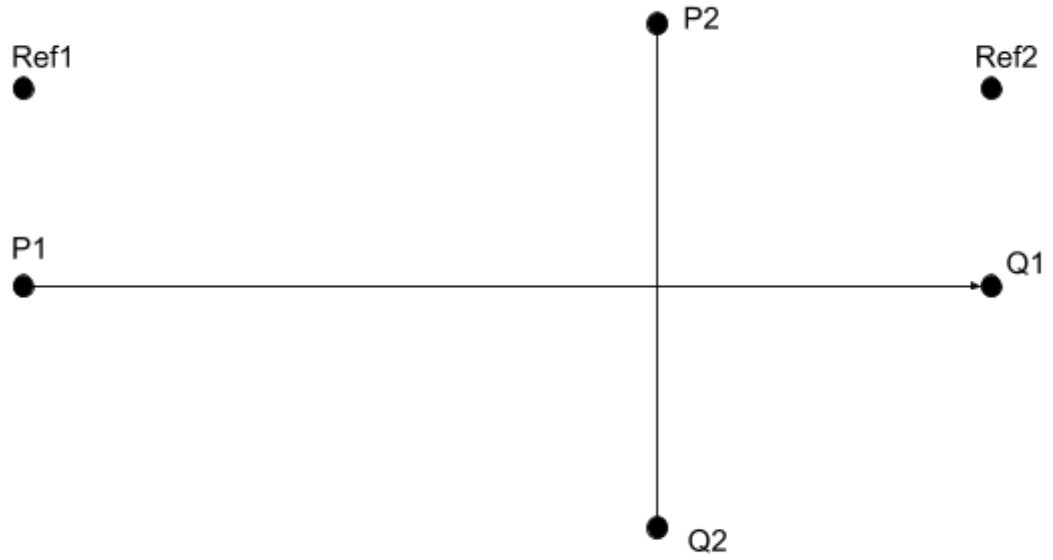
Per cada triangle hem de cridar aquestes funcions tres vegades, per comprovar les tres línies que el formen.

```
for (int i = 0; i < outer_hull.size(); i++)
{
  init_point = outer_hull[i] + initial_position;
  end_point = outer_hull[i] + initial_position + move_vec;

  for (std::vector<Triangle>::iterator tr = solid_list.begin(); tr != solid_list.end(); tr++) {
      if (glm::length(move_vec) > 0 && Physics::doIntersect(init_point, end_point, flatter_center,
tr->point_1, tr->point_2, tr->point_3)) {
         move_vec = Physics::processIntersection(init_point, end_point, tr->point_1, tr->point_2);
         end_point = outer_hull[i] + initial_position + move_vec;
      }

      if (glm::length(move_vec) > 0 && Physics::doIntersect(init_point, end_point, flatter_center,
tr->point_2, tr->point_3, tr->point_1)) {
         move_vec = Physics::processIntersection(init_point, end_point, tr->point_2, tr->point_3);
         end_point = outer_hull[i] + initial_position + move_vec;
      }

      if (glm::length(move_vec) > 0 && Physics::doIntersect(init_point, end_point, flatter_center,
tr->point_3, tr->point_1, tr->point_2)) {
          move_vec = Physics::processIntersection(init_point, end_point, tr->point_3, tr->point_1);
          end_point = outer_hull[i] + initial_position + move_vec;
      }
}
```

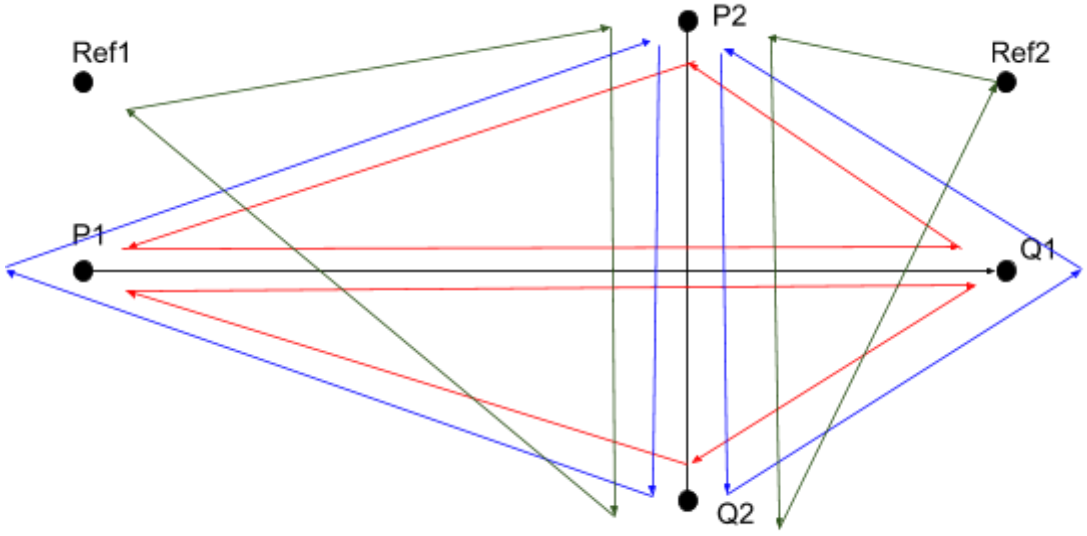
La primera d'aquestes funcions, `doIntersect`, rep dos conjunts de tres punts que formen el punt inicial i final del moviment i la banda on hi ha el sòlid de l'objecte i el punt inicial i final de la paret i altra vegada, el punt on la paret és sòlida:

```
bool Physics::doIntersect(glm::vec2 p1, glm::vec2 q1, glm::vec2 ref1, glm::vec2 p2, glm::vec2 q2,
glm::vec2 ref2, bool test)
```

Per exemple, els punts que rebria la funció serien els següents:



Amb aquests punts es calcula si intersecciones utilitzant la orientació entre ells:



El conjunt de fletxes indiquen la orientació dels punts, ja sigui horària, antihoraria o colineals. Les fletxes blaves indiquen la orientació dels punts P1P2Q2 i Q1P2Q2. Podem veure que el primer gira en sentit horari i el segon en sentit antihorari, cosa que ens indica que els punts P1 i Q1 son en bandes oposades del segment P2Q2. De la mateixa manera, les fletxes vermelles indiquen la orientació dels punts P2P1Q1 i Q2P1Q1. Altra vegada podem veure que giren en sentits diferents, confirmant que existeix la colisió.

Per assegurar-nos que el xoc existeix, hem de comprovar que la part sòlida dels dos elements no sigui a la mateixa banda, cosa que indicaria que el punt està “sortint” de dins d’un sòlid. Per això utilitzem les fletxes verdes, que calculen l’orientació de P2Q2Ref1 i P2Q2Ref2. Si segueixen sent contrarotatòries confirmarem que hi ha colisió.

```
int o1 = orientation(p1, q1, p2);
int o2 = orientation(p1, q1, q2);
int o3 = orientation(p2, q2, p1);
int o4 = orientation(p2, q2, q1);

int o5 = orientation(p2, q2, ref1);
int o6 = orientation(p2, q2, ref2);
```

La funció `orientation` ens indicarà l'orientació dels punts: 1 si giren en sentit horari, 2 si es en sentit antihorari o 0 si son colineals. El cas general que s'ha de validar correspon a la següent condició:

```
if (o1 != 0 &&  //p1, q1, p2 no son colineals
    o2 != 0 &&  //p1, q1, q2 no son colineals
    (o3 == 0 && o6 == o4 || o3 != 0) &&  //p2, q2, p1 no son colineals o si ho son, el moviment es
en el sentit de ref2
    o1 != o2 && //p2 i q2 son en bandes oposades de p1q1
    o3 != o4 && //p1 i q1 son en bandes oposades de p2q2
     o5 != o6 && //ref1 i ref2 son en bandes oposades de p2q2
     o3 != o6 //ref1 i ref2 son en bandes oposades de p2q2
     ) {
       return true; //true
}
```

Si aquesta condició es compleix es retornarà cert i el joc seguirà calculant el punt de la colisió. Si no es compleix cal comprovar que no es tracti d'un cas especial:

```
// p2, q2 and q1 are colinear and q1 lies on segment p2q2
if (o4 == 0 && o5 == o6 && onSegment(p2, q1, q2)) {
     return true;  //true
}
```

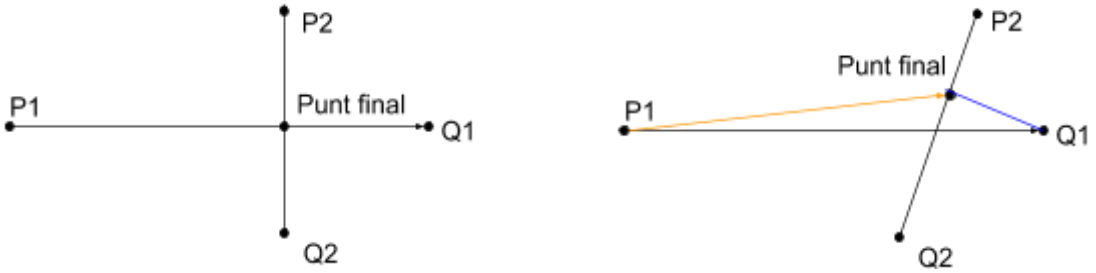
En aquest cas, és possible que el moviment sigui paral·lel a paret, de manera que tot i que el punt final està tocant la línia, no hi hagi col·lisió. Hem de comprovar que si es dóna aquest cas les referències dels sòlids siguin al mateix costat, cosa que significa que el planar es troba en una cantonada.

Per últim, en el cas que hi hagi col·lisió calcularem el punt exacte:

```
glm::vec2 Physics::processIntersection(glm::vec2 p, glm::vec2 q, glm::vec2 r, glm::vec2 s) {

       glm::vec2 pq, rs, sr;
       pq = q - p;
       rs = s - r;
       sr = s - r;

       float dot, det, angle_pqrs, angle_pqsr;
       dot = pq.x*rs.x + pq.y*rs.y;
```

```
        det = pq.x*rs.y - pq.y*rs.x;
        angle_pqrs = atan2(det, dot);

        //Les dues línies son pràcticament paraleles
        if (abs(abs(angle_pqrs) - MathHelper::PI) < 0.001 || abs(angle_pqrs) < 0.001) {
        if (glm::length(p - r) > glm::length(p - s)) {
                return s - p;
        }
        else {
                return r - p;
        }
        }
        else {
        glm::vec2 meet_point = Physics::lineLineIntersection(p, q, r, s);

        glm::vec2 a, b;
        a = q - meet_point;
        b = s - r;

        float proj = ((a.x * b.x + a.y * b.y) / glm::length(b));
        glm::vec2 result = glm::normalize(b) * proj;

        glm::vec2 res = (meet_point + result) - p;

        return res;
        }
}
```

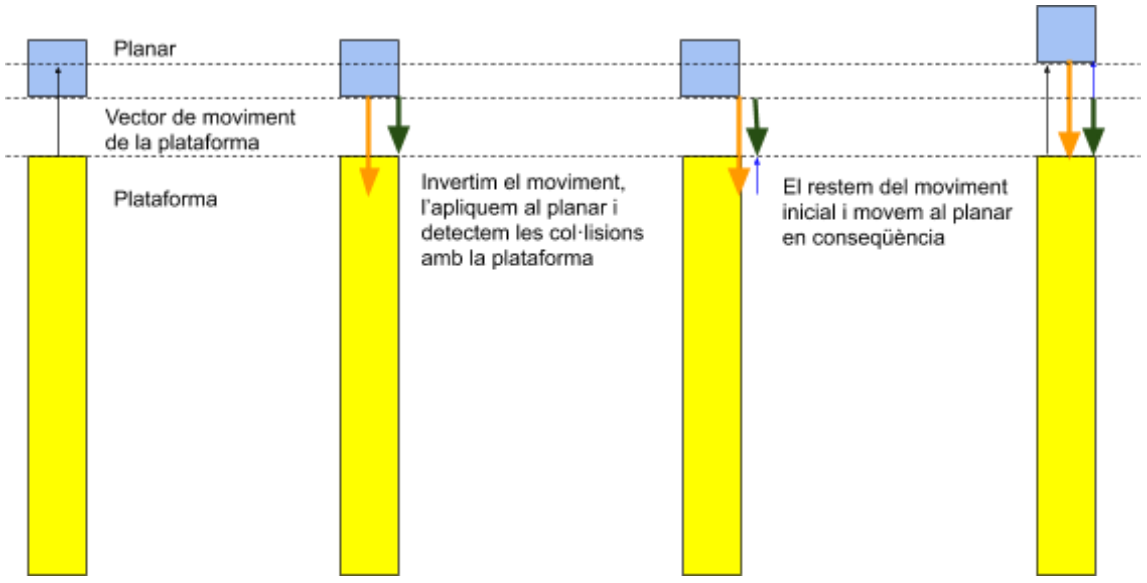
Aquesta funció calcularà el punt en què els segments pq i rs es toquen i tindrà en compte la projecció de pq sobre rs per simular que el punt es mou fregant la paret.



Tornant a la funció inicial, havent obtingut el nou punt de xoc es recalcularà el vector de moviment i es seguirà amb la iteració de la resta de polígons.

Finalment, per calcular el l'afectació del moviment de les plataformes es farà servir el seu vector de moviment que hem calculat prèviament. Degut a la manera com funcionen les col·lisions, moure la plataforma no serviria per comprovar si xoca amb el planar, de manera que el que farem serà invertir el moviment de la plataforma i aplicar-lo al planar. Recalcularem les col·lisions,

aquesta vegada només cal amb la pròpia plataforma i el vector que ens resulti s'haurà de restar al moviment original de la plataforma i aplicar-lo al planar:



Per últim, calculem les col·lisions amb les superfícies de mort i si detectem que el planar les està tocant li canviem l'estat a mort perquè la resta del joc ho tingui en compte.

En el cas del jugador les col·lisions es calculen de la mateixa manera, duplicant l'instant on es troba el jugador i substituïnt les dades del planar per les del jugador. D'aquesta manera, l'instant que ens retorna el mòdul de físiques conté en les dades del planar l'estat final del jugador. Aquestes dades son extretes i aplicades al jugador.

3. Conclusions

La conclusió principal del treball es que si, es pot desenvolupar un videojoc funcional que exploti la idea de la informació meta-perfecte. S'han plantejat mecàniques que explotarien aquesta idea i dotarien al joc de la seva aura única i nova. Nova perquè en els estudis de mercat que s'han fet no s'ha trobat cap joc similar; el més semblant consisteix en els histogrames sonors en 3D, que representen el temps com a espai.

Un cop acabat el treball, cal dir que no s'han assolit els objectius que es plantejaven a l'inici, tot i que si que s'ha pogut explorar en profunditat la idea de la informació meta-perfecte. Els requisits necessàris per crear un joc atractiu i interessant hi son presents però les limitacions tècniques i temporals han fet impossible desenvolupar-lo adequadament.

Tot i amb això el producte obtingut no és inútil, serviria com a fonament si es continués desenvolupant aquesta idea i serviria com a framework per experimentar noves idees o interaccions.

La feina que s'ha realitzat s'ha ajustat adequadament a la planificació tot i que la naturalesa investigativa del treball ha acabat alterant les dates previstes inicialment, que al seu torn van donar problemes a l'hora d'entregar les PAC's.

En un futur, com s'ha comentat, es podria fer servir el producte d'aquest treball com a prototip per desenvolupar un videojoc complet.

4. Glossari

- Jugador: Està compost per la càmera que controla l'usuari. Està sotmès a les físiques i col·lisions i pot donar ordres al planar.

- Planar: Criatura tridimensional (2 espacials i 1 temporal). Es tracta de la criatura a la que el jugador pot donar ordres.

- FPS: Frames per second. Correspon a la quantitat d'imatges per segon que es mostra per pantalla.

- Instant: Cada escena es divideix en múltiples instants que son cada una de les mostres de temps discretes del nivell i el planar.

5. Bibliografia

- https://learnopengl.com/, Febrer 2019 a Juny 2019
- https://stackoverflow.com/, Febrer 2019 a Juny 2019
- https://www.opengl.org/, Febrer 2019 a Juny 2019
- https://glm.g-truc.net/0.9.9/index.html, Febrer 2019 a Juny 2019
- https://www.glfw.org/, Febrer 2019 a Juny 2019
- https://www.freetype.org/, Febrer 2019 a Juny 2019
- https://www.youtube.com/channel/UCqJ-Xo29CKyLTjn6z2XwYAw, Febrer 2019 a Juny 2019
- https://en.wikipedia.org/wiki/Perfect_information, Febrer 2019
- https://policonomics.com/lp-game-theory1-perfect-imperfect-information/
- http://econ.ucsb.edu/~garratt/Econ171/Lect14_Slides.pdf, Febrer 2019
- https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/, Abril 2019
- https://en.wikipedia.org/wiki/Distance_from_a_point_to_a_line,Abril 2019