234326
Computer Graphics Project
Platon
Game Project

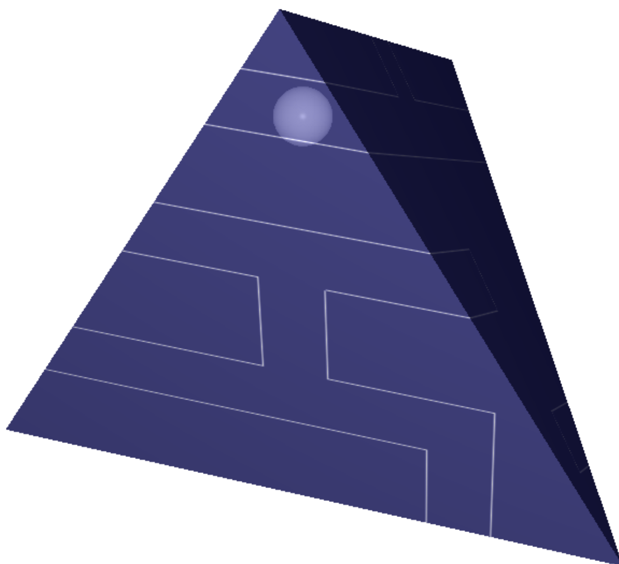
Student: David Keter
Advisor: Mr. Roi Poranne

Introduction

Computer games play an important role in today's culture and society; and with the fast advancements in hardware and technology came more possibilities and higher user demand for better games, both visually appealing and in gameplay.

The most basic functionalities needed to make a game are the rendering engine and physics engine, and in this project, I attempt to build these engines with help of hardware acceleration rendering to create a user-friendly 2D game in a more complex 3D environment.

Game Description



The game is a puzzle platformer, with a simple 2 dimensional movement. The stage however, is mapped onto different 3D objects giving an additional element to the game. Moving from one face to the other causes the shape to rotate making the downward gravitation to change direction and thus allowing access to new areas.

The goal of the game is to find the exit in each stage by making use of this 3D feature.

Game Physics

To allow the player to move around and fall from gravity, the game must have an appropriate physics engine. The engine works by solving frame by frame independently, and should manage collision detection and response of the moving player, while supporting player rotation in order to move from one face of the shape to another.

Core Concepts

Entity:

Entity is a 2D geometrical convex shape, used as a bounding shape of objects for collision detection. It also holds a position vector, velocity vector and rotation.

You can create an entity by passing a convex shape, defined by a vector of points in counter-clockwise winding (CCW), and the segments connecting every two points should not intercept.

```
intern::Convex shape;
shape.vertices.push_back(Point(0,0));
shape.vertices.push_back(Point(10,0));
shape.vertices.push_back(Point(10,10));
shape.vertices.push_back(Point(0,10));
Entity entity(shape);
```

The entity also has some methods to allow the user to manually change the velocity and rotation values.

Map Partition:

As will be explained further down in this document, we subdivide each stage into smaller partitions per face on the 3D shape. Each map partition holds a set of line segments for entities to interact with.

To create a map partition, a unique id is required to access it. Line segments are added one after another by providing both endpoints. When finished, the map must preprocess the lines to create a grid (explained later on) so we must call the `initialize()` function at the end.

```
MapPart face(12345); // our unique id
face.line(Point(0,0),Point(-50,20));
face.line(Point(-50,20),Point(100,80));
face.initialize();
```

Unlike the convex shape in entities, there are no restrictions on intersection between line segments.

After we have an entity and a map part, we can place or erase the entity in it by calling:

```
void insert( Entity& character, const Point& origin, double angle );
void erase( Entity& character );
```

These functions create an instance of the entity in the map part, so having the same entity in multiple map parts is a supported feature.

To make the entity automatically move from one map part to another, create a warp line before initializing the map.

```
void warp( Point p0, Point p1, uint32 targetMap, Point t0, Point t1 );
```

The parameters are two points just like a regular line, then a target map part id number, and how the same line would be placed in the target map.

World:

The physics world is a collection of entities and map parts that interact together. The world class is used to advance time and solve collisions between entities and map lines.

The world class is a static singleton and does not need to be created. When creating entities and map parts, they are automatically added to the world collection. To advance one time frame, simply call

```
World::solve();
```

Rules:

To let the world know what operations to do on collision events, we can define methods classified as rules that the world solver will call upon when needed. Lines are subdivided into two types: pierceable (such as warp lines) and non-pierceable (regular lines). The world solver knows the type of line by calling `Rules::pierce()`. And different rule methods apply for each type:

- Pierceable: beginContact/overContact/endContact are called when the entity begins touching the line, each time frame that it's colliding with it, and when it's no longer intersecting respectively.
- Non-pierceable: contact is called when a collision is detected; touchContact is for collisions by rotation.

To set our custom rules class, we call:

```
World::setRules( Rules& rules );
```

And the referenced rules class should not be deallocated while solve() is still being called.

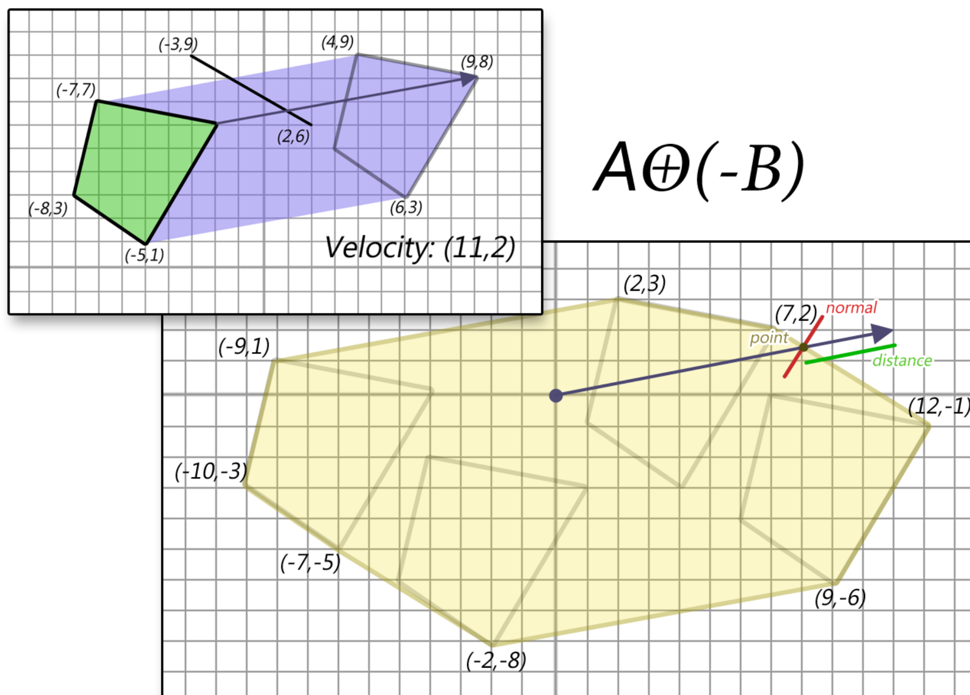
Collision Detection

Given a convex shape with position and velocity and a line segment, we want to check whether the shape hits the line at some point in its path, and if it does, find the collision distance, collision point and normal.

The solution I used to solve this problem is similar to the Gilbert-Johnson-Keerthi distance algorithm[i] in the fact that it involves using the Minkowski Sum.

Minkowski Sum is defined as $A \oplus B = \{a + b | a \in A, b \in B\}$

It can be proved that two convex shapes *A* and *B* are intersecting iff the convex hull of $A \oplus (-B)$ contains the origin. With this in mind, it isn't difficult to detect a collision between a moving convex shape and line segment by checking if the origin is inside the convex hull of the Minkowski sum of the convex shape, stretched by the velocity, with the negated line segment. The collision distance can then be computed from the velocity vector minus the penetration vector, and the normal and collision point are computed by the intersection of the velocity vector starting from the origin and the segment on the convex hull.



Visual example of how to compute collision values

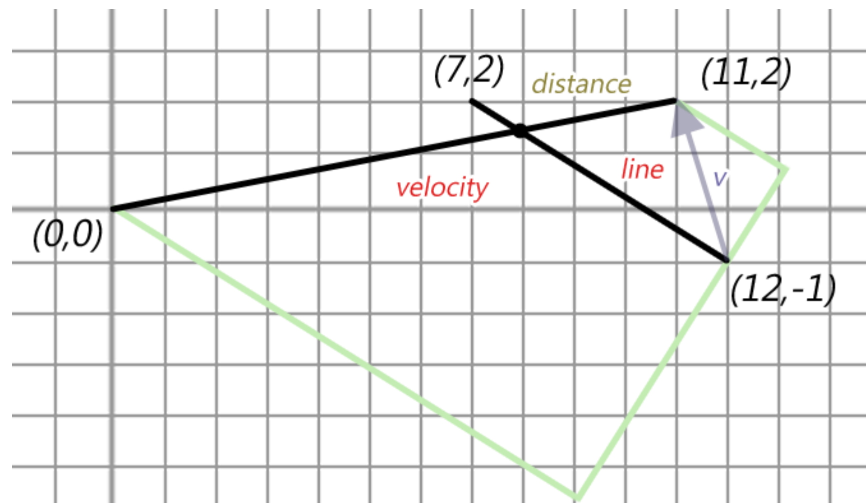
Notes:

- The original GJK algorithm is used only for computing distance between two convex shapes, and does this by attempting to find a triangle in where the origin is located in, each time updating the furthest vertex. The GJK algorithm converges fast and is a popular solution for discrete collision detection. In our case however, we want to solve continuous collisions and therefore also need information such as the collision distance in the velocity direction and normal.
- The collision point is computed by storing the source values of all the vertices before doing the Minkowski sum so that we can return the point to its correct value in the original coordinates.
- In practice, we don’t need to compute all points in the Minkowski sum; by taking the vertex furthest in a given direction on both shapes, we can be sure to get a point in the convex hull boundary. So it’s possible to iterate through the convex hull without computing interior points.

After we know the collision distance, point and normal, we can simply project the remaining velocity after the collision towards one of the sides, depending on the normal and repeat the process until all the velocity has been used in that frame.

Numerical Errors:

Mathematical computations are made using 32 bit integer values; vertex coordinates and movement vector are stored as whole integer values, and operations such as inner and cross product produce 64 bit values so that we do not lose information. The reason behind this is because numeric errors in computing the distance play an important role in the correctness of the solution. If by error, the object is placed overlapping the line, then it will affect the next collision detections, regardless of the moving direction. We solve by noticing how the distance can also be represented exactly by a fraction:



$$\frac{|distance|}{|velocity|} = \frac{|\vec{v} \times \widehat{line}|}{|\overrightarrow{velocity} \times \widehat{line}|} = \frac{|\vec{v} \times \overrightarrow{line}|}{|\overrightarrow{velocity} \times \overrightarrow{line}|}$$

The fraction then tells with precision, how much of the velocity we can use before the collision. The fraction holds two 64 bit integer values for the numerator and denominator values; and fraction operations such as relative comparisons make internal use of a 128 bit integer value. The only rounding we do is in the resulting velocity before collision, so that the new position coordinates stay whole, and by rounding it down to whole integer values, we can be sure to be on the safe side of the line.

This method allows for less “patches” when placing objects arising from floating point math, such as the popular margin area, or “skin”, around a shape used in public physic engine libraries such as Box2D or Chipmunk physics[ii].

Grid optimization

To not check collision with all line segments on the map, we hold a small region around the player with a small subset of line segments to check with. This can be done by subdividing the map into an irregular sized grid where each cell holds a list of segments that pass through that the cell’s area; the grid subdivision is defined by the projection of the endpoints onto the y and x axis. The region then holds a square area of cells, and we update the region by adding/removing cells from the square from all sides according to the new position and velocity values.

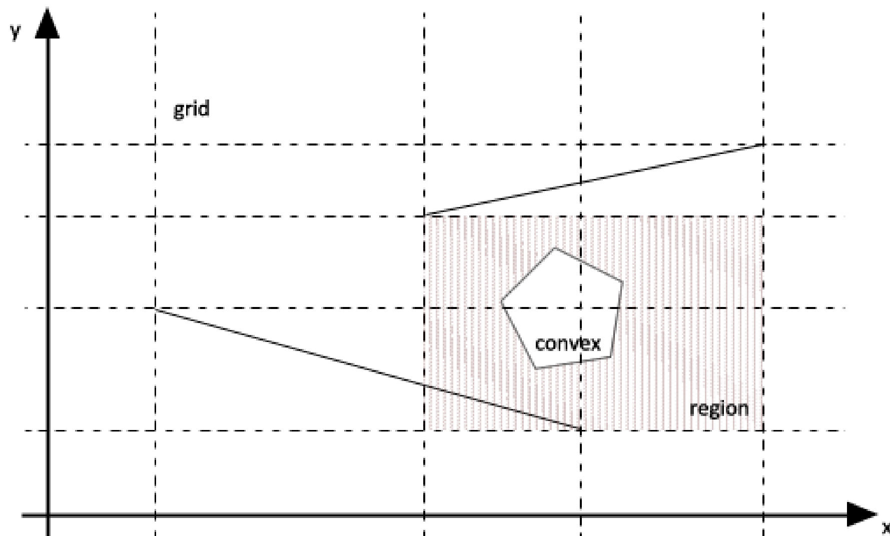


Image showing a region and map grid example

Face transitions and multiple player instances

When the player moves from one face of the 3D shape to another the player shape and orientation rotates, and the player's position might suddenly jumps from one spot to another due to the need of mapping a 2D set of lines onto a 3D surface. For simplicity, instead of holding all lines in a single 2D space, we define a set of lines per face, independent of other faces. Then when moving from one face to another, we insert an instance of the player shape into the new face, and when leaving a face, we erase the previous instance.

By allowing multiple simultaneous instances of the player's shape, we can avoid errors in the transition between faces, and assert correctness. However, with multiple instances, collision detection must be synced between instances so that hitting a line on one face also affects the instance on the other face. Each instance holds a local orientation and position offset from the parent shape, which is computed when entering a new face depending on the face edges; collisions are computed relative to the instance, but the distance is applied equally to the parent shape, thus affecting all instances.

Game Visual

In practice, each face of the 3D shape holds its own 2D lines for the game physics to compute. These lines are transformed into 3D space by a matrix transformation constructed from the face coordinates before rendering.

To prevent two faces overlapping each other and obstructing the camera's viewpoint, we assume the 3D shapes are convex, so that if we place the camera on the straight line coming from the shape's center and piercing through the center of the player, the player will always be visible. For smooth camera rotation as we move around, we use quaternion math.

For simplicity, we pick all edges to be of the same size, thus the only shapes used are the platonic solid shapes[iii]; the name of the game, Platon, comes from this.

The 5 Platonic Solids (images from Wikipedia)

Core Concepts

Player:

The player class implements the user interaction within the game. It derives from the physical entity class and as such it takes care of pressed keys in order to move the player entity in the map. In our case, the player automatically builds the entity shape as a small sphere.

Player has two fundamental methods:

```
void tick( rsys::Program& program );
```

which is used to process the input and update the player's velocity, depending on the situation, and

```
void draw( graphics::Display& display );
```

used to render a small sphere to the screen at the current position.

Camera:

A specialized camera class is also defined to track the player position and orientation. It also has two methods:

```
void set( const Player& player, PlatonicSolid& stage, uint32 w, uint32 h );
void lerp( const Player& player, PlatonicSolid& stage );
```

The first is to place the camera to look at the player on the given stage, and should be called at initialization. The second method is to interpolate between the previous and next camera position to give a smooth transition.

Platonic Solid:

This object holds a set of faces, each one with its own physical map partition. The platonic solid is loaded from a file and simply needs to be drawn to the screen with

```
void draw( graphics::Display& display );
```

The file format is as follows

- First line declares the type of solid:
  TETRAHEDRON | HEXAHEDRON | OCTAHEDRON | DODECAHEDRON | ICOSAHEDRON
- Second line holds the initial face index, position and rotation to insert the player
- Third line is for the goal coordinates
- Fourth line holds the face size used to build the shape

Then each face is described in order

- Starting with a % for every face
- Each line holds the map segment end point parameters: point0x point0y point1x point1y

Sources and Reference

[i] GJK in Collision Detection, Game Development by William (http://www.codezealot.org/archives/88)
[ii] Box2D Documentation (http://www.box2d.org/manual.html#_Toc258082970)
[iii] Platonic Solids, in Math is Fun (http://www.mathsisfun.com/platonic_solids.html)