A Framework for Developing Android Games

Tobias Morgan
1643182
BSc Mathematics and Computer Science
Supervised by Shan He

School of Computer Science
University of Birmingham
April 2019

Synopsis

In recent years, the Android mobile platform has been rapidly rising in popularity and with it comes an increasing number of video games being released for the platform every month.
This report will detail the design and implementation of a framework designed for developing 2D Android games. It will provide algorithms for collision detection and resolution, a set of AI systems (including behaviour trees, a navigation mesh generator, and path-finding algorithm), along with numerous other time-saving and convenience modules.
The framework is not intended to be a full game engine akin to Unity or Unreal, as it does not provide a development environment which the user can interact with, via a graphical user interface. What it is intended to do however, is to supply a core set of tools that are common to most video games so that the user can focus more on the creative aspect of game development.
To help show that the framework streamlines the development process, this report will also detail the development of a 2D top-down dual-stick shooter Tank game in which, the player will take control of a tank, and be tasked with defeating waves of enemy tanks.

Acknowledgements

I would like to thank my friends and family for helping me over the last few months, and thank you to my supervisor Shan He, for providing support throughout the course of this project, as well as the inspectors from both of my project demonstrations for providing valuable feedback on how to progress with both the project, and this report.

All Code for this project can be found at: https://git-teaching.cs.bham.ac.uk/mod-ug-proj-2018/txm683/tree/master Main code package is at : .../master/app/src/main/java/bham/student/txm683/heartbreaker

Contents

Chapter 1

Introduction

1.1 Introduction

In recent years, Android has become the most popular mobile operating system [1]. This increase in popularity has lead to an increase in the Android video game market. These days, most games released on the platform are created using a game engine, for example: Unity or Unreal. These pieces of software provide a graphical environment which is used to develop 2D and 3D games. However, these engines have a steep learning curve and it's rare that a game will need all of the features that the engine can offer.

If you want to create a 2D arcade game, these engines can be overkill as they are using algorithms and approaches designed for AAA titles and 3D environments. This raises the need for a game framework, instead of a full blown game engine. A game framework contains a lot of the important algorithms that games need to function, for example collision detection algorithms and path-finding techniques, the implementations of which are consistent across most games.

The aim of this report is to outline the development of such a framework. It will describe the algorithms used and their benefits and drawbacks. The second half of this report will detail the development of a game called Tanktrum, which is a 2D top-down shooter. This second section will cover the aspects of a game that are not covered in the framework, for example, the game loop and the rendering engine.

1.1.1 The Report Files

The zip file included with this report contains the full directory of code for this project, along with a file containing the address of the git repository that this code can also be found.

1.1.2 Running the Program

In order to run this program, you will need to download Android Studio or Intellij, and the Android SDK. Using either of these programs, you will need to import the project from the git repository and it will load all the file structures for you. Both of these platforms offer an emulated Android phone which you can run the software on, or you can run it onto an android phone which is connected via USB. Please note that if you are running the project on an emulator, the performance will not properly reflect that of the game, I recommend running it on a phone. This article will help you to run the project: https://www.jetbrains.com/help/idea/running-and-debugging-android-applications.html namely the first section on running an Android program.

1.2 Background Knowledge

1.2.1 Axis-Aligned Bounding-Box

An Axis-Aligned Bounding-Box (AABB) is a type of bounding volume for a shape. It is defined by four values: the maximum and minimum x and y values of a shape's vertices. See figure 1.1 for a visual aid.

Checking if two AABBs intersect, is simply 4 boolean checks and is therefore a cheap operation to perform, as can be seen in listing 1.1.

Listing 1.1: Intersection methods in the BoundingBox class

```
public boolean intersecting(BoundingBox bb){
    return this.left < bb.right && this.right > bb.left &&
            this.top < bb.bottom && this.bottom > bb.top;
}
```

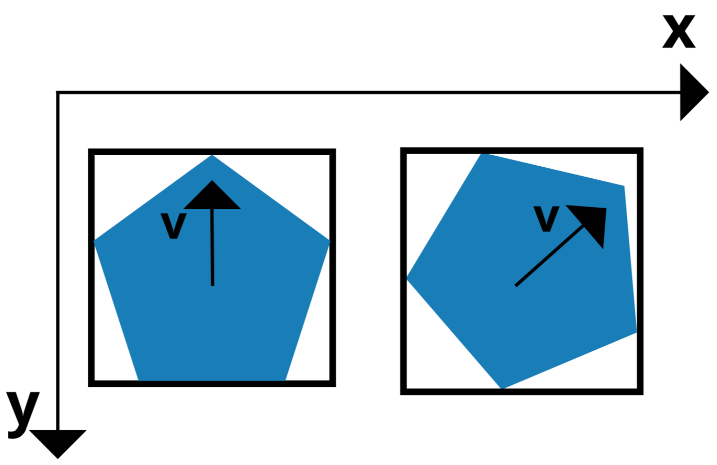


Figure 1.1: An example of an AABB, with the vector v representing the shape's forward vector.

```
public boolean intersecting(Point p){
    return this.left < p.getX() && this.right > p.getX() &&
           this.top < p.getY() && this.bottom > p.getY();
}
```

Chapter 2

Developing The Framework

2.1 Specification

The framework should include the following functionality:

1. Collision detection system
2. AI path-finding system
3. AI behaviour tree system
4. Navigation mesh generation algorithm
5. User interface elements
6. Level creation system

Collision Detection

The collision detection part of this framework should fulfill the following requirements:

- Provide a set of algorithms for detecting collisions between objects in the game world
- Collision detection tools will be loosely coupled to allow the user to choose which method to use.
- An interface will allow the user to call the methods in the framework with their own custom objects

AI Path-finding

The path-finding part of this framework should fulfill these requirements:

- A graph class for use with A*
- The A* algorithm for plotting paths
- The graph class should be able to be used by the user for other purposes

AI Behaviour Tree System

In order for the user to implement complex AI behaviours, the following requirements should be met:

- Support for composite, decorator, conditional nodes should be present
- The user should be able to define their own nodes
- The user should be able to use pre-defined behaviour trees as nodes in their own trees
- example behaviour trees should be provided

Navigation Mesh Generation

The framework needs an inbuilt system for producing a graph for the path-finding system to use, as well as a system of obstacle avoidance which meet the following requirements:

- AI should be able to move from one node to it's neighbour without colliding with another game object
- A graph should be produced that can be used with the A* algorithm.
- The navigation mesh should be partitioned every game tick so that any entities that move are removed from the mesh.

User interface elements

The framework does not include a rendering engine however, the following requirements should be met regarding user interfaces:

- Game objects should have a draw method targeting the Android framework's Canvas API
- A system for displaying messages to the player should be in place.
- An interface containing methods enabling the use of Android canvas to draw custom objects.

Level creation system

The user should be able to create levels using tools provided in this framework with the following requirements:

- Maps should be designed in a tile based fashion
- Movement should be continuous, i.e not confined to a tile based system.
- A separate program should be created that the user can draw out maps
- The created maps should be read in via an image processing algorithm

2.1.1 Constraints

- The framework must be developed in Java
- The framework must be developed for the Android Platform

2.1.2 Dependencies

- org.json for parsing objects from json files
- HashCodeBuilder from apache.commons for creating hashable objects.
- The Android SDK to allow the framework to use Android elements.

2.2 Tile, Point & Vector

2.2.1 Introduction

The following subsections describe helper classes for expressing different forms of coordinates. They are all immutable, and any manipulation functions their classes contain will not modify the object, instead returning the result as a new object. I chose to do this because most of the operations I use these classes for, I don't want to modify the original value, for example interpolating the velocity for some time step shouldn't affect the stored value of velocity. // All of these classes override equals and hashcode so that I can use them in HashMaps and HashSets for instant access. The hashcode implementations used apache commons hashcodebuilder which can be found at: https://commons.apache.org/proper/commons-lang/javadocs/api-3.1/org/apache/commons/lang3/builder/HashCodeBuilder.html.

2.2.2 Tile

Tile is a helper class to easily express a cell coordinate without having to create two integers every time I want to reference a cell.

Listing 2.1: Tile fields and constructors

```
private final int x;
private final int y;

//most commonly used constructor
public Tile(int x, int y){
    this.x = x;
    this.y = y;
}

//used to allow easy conversions between the float values of Point and the
//int values of Tile
public Tile(Point point){
    this.x = (int) point.getX();
    this.y = (int) point.getY();
}

//used when loading the spawn locations of certain objects from
//a level json file.
public Tile(JSONObject jsonObject) throws JSONException {
    this.x = jsonObject.getInt("x");
    this.y = jsonObject.getInt("y");
}
```

2.2.3 Point

This class is mainly used to express the positions of items in world coordinates, or as screen coordinates. The reason for them being implemented as floats, is that the Android rendering canvas takes floats for it's positions, and it means that I don't have to cast the objects every time that I want to interact with an Android system. The implementations aren't all too different from the Tile implementations, except the equals method includes a delta allowance to account for floating point precision errors.

Listing 2.2: Point equals() Method

```
@Override
public boolean equals(@Nullable Object obj) {
    if (obj == this){
        return true;
    }

    if (!(obj instanceof Point)){
        return false;
    }

    Point p = (Point) obj;

    float delta = 0.0001f;

    return (Math.abs(p.x-x)<delta) && (Math.abs(p.y-y)<delta);
}
```

2.2.4 Vector

The Vector class is one of the most critical classes in this framework. Without a proper implementation, the collision detection, movement input, and other classes would be a lot more awkward to implement. The more complex methods of this class have been unit tested to ensure they provide the correct behaviour.

This implementation includes both affine space and vector space qualities for ease of use. In an affine space, a vector is defined as a line between two points in space, whereas a vector in a vector space is defined by it's magnitude and direction.

Each vector stores the points of the head and the tail upon creation, and also the relative distances between the head and tail.
The points of the head and tail are used for affine operations like translate. The relative distances are used for vector operations like dot, angle between.

Listing 2.3: Vector fields and constructors

```
//static constant for zero vector to save mutiple heap allocations
//for the same thing.
public static final Vector ZERO_VECTOR;

static {
    ZERO_VECTOR = new Vector();
}
//affine space definition
private final Point tail;
private final Point head;

//vector space definition
private final float xRelativeToTail;
private final float yRelativeToTail;

public Vector(){
    tail = new Point();
    head = new Point();

    xRelativeToTail = 0f;
    yRelativeToTail = 0f;
}

/**
 * Constructs a new Vector object with tail at the origin and head at
 * the Point constructed
 * from the given components.
 *
 * @param headX x component of new vector relative to origin
 * @param headY y component of new vector relative to origin
 */
public Vector(float headX, float headY){
    tail = new Point();
    head = new Point(headX, headY);

    xRelativeToTail = headX;
    yRelativeToTail = headY;
}

/**
 * Constructs a new Vector object with tail at the origin and head
 * at the given Point.
 * @param head Point to place head of vector
 */
public Vector(Point head){
    this(head.getX(), head.getY());
}

/**
 * Constructs a new Vector object with tail at the given position and
 * head at the given position.
 * @param tail Position of the new vector's tail
 * @param head Position of the new vector's head
 */
public Vector(Point tail, Point head){
    this.tail = tail;
    this.head = head;

    xRelativeToTail = head.getX() - tail.getX();
    yRelativeToTail = head.getY() - tail.getY();
}

public Vector(JSONObject jsonObject) throws JSONException{
    this(new Point((JSONObject)jsonObject.get("tail")),
            new Point((JSONObject)jsonObject.get("head")));
}
```

Supported Operations

A list of the main operations with implemented method names:

- Get vector's length: getLength
- Set vector's length: setLength
- Normalize the vector: getUnitVector
- Vector addition: vAdd
- Translate Vector: translate
- Scalar Multiplication: sMult
- Dot product: dot
- Cross product: det
- Angle between two vectors (Listing 2.4): Vector.calculateAngleBetweenVectors
- Rotate a vector (Listing 2.5): rotate

Listing 2.4: Implementation ofcalculateAngleBetweenVectors

```
public static float calculateAngleBetweenVectors(Vector primaryVector,
            Vector movementVector){
    //calculates the angle FROM the primaryVector TO the movementVector
    primaryVector = primaryVector.getUnitVector();
    movementVector = movementVector.getUnitVector();

    float dot = primaryVector.dot(movementVector);
    float det = primaryVector.det(movementVector);

    return (float) Math.atan2(det, dot);
}
```

Listing 2.5: Implementation of Vector.rotate()

```
/**
* Rotates this vector by an angle. This vector's tail position is used
* as the origin of rotation
* @param cosAngle Cos of angle to rotate by
* @param sinAngle Sin of angle to rotate by
* @return The rotated vector.
*/
public Vector rotate(float cosAngle, float sinAngle){
    float newX = xRelativeToTail * cosAngle - yRelativeToTail * sinAngle;
    float newY = xRelativeToTail * sinAngle + yRelativeToTail * cosAngle;
    return new Vector(tail, new Point(newX + tail.getX(), newY + tail.getY()));
}
```

Unit Testing

As this class is an important one, the core methods have been unit tested to ensure the correct behaviour is occurring.

Listing 2.6: Unit Testing for the Vector class

```
private static final float DELTA = 0.0001f;
private static final float ROOT2 = (float) Math.sqrt(2f);

private static final float FX = -1.7445f;
private static final float FY = 234.7654f;
private static final float FLENGTH = 234.77188f;

private static final float GX = 1234.2345f;
private static final float GY = 234.1234f;
private static final float GLENGTH = 1256.2438f;

private Vector a = Vector.ZERO_VECTOR;
private Vector b = new Vector(-1,1);
private Vector c = new Vector(1,-1);
private Vector d = new Vector(1,1);
private Vector e = new Vector(-1,-1);
private Vector f = new Vector(FX, FY);
private Vector g = new Vector(GX, GY);

private Vector h = new Vector(new Point(1,2), new Point(2,3));
private Vector i = new Vector(new Point(3,1), new Point(2,2));
private Vector hAddi = new Vector(new Point(1,2), new Point(1,4));

@Before
public void setUp() throws Exception {
    a = Vector.ZERO_VECTOR;
    b = new Vector(-1,1);
    c = new Vector(1,-1);
    d = new Vector(1,1);
    e = new Vector(-1,-1);
    f = new Vector(FX, FY);
    g = new Vector(GX, GY);
}

@Test
public void getUnitVector() {
    Vector ua = Vector.ZERO_VECTOR;
    Vector ub = new Vector(-1/ROOT2,1/ROOT2);
    Vector uc = new Vector(1/ROOT2,-1/ROOT2);
    Vector ud = new Vector(1/ROOT2,1/ROOT2);
    Vector ue = new Vector(-1/ROOT2,-1/ROOT2);
    Vector uf = new Vector(FX/FLENGTH, FY/FLENGTH);
    Vector ug = new Vector(GX/GLENGTH, GY/GLENGTH);

    assertTrue(ua.equals(a.getUnitVector()));
    assertTrue(ub.equals(b.getUnitVector()));
    assertTrue(uc.equals(c.getUnitVector()));
    assertTrue(ud.equals(d.getUnitVector()));
    assertTrue(ue.equals(e.getUnitVector()));
    assertTrue(uf.equals(f.getUnitVector()));
    assertTrue(ug.equals(g.getUnitVector()));
    assertEquals(1f, b.getUnitVector().getLength(), DELTA);
    assertEquals(1f, c.getUnitVector().getLength(), DELTA);
    assertEquals(1f, d.getUnitVector().getLength(), DELTA);
    assertEquals(1f, e.getUnitVector().getLength(), DELTA);
    assertEquals(1f, f.getUnitVector().getLength(), DELTA);
    assertEquals(1f, g.getUnitVector().getLength(), DELTA);
}

@Test
public void getLength() {
    assertEquals(0f,a.getLength(), DELTA);
    assertEquals(ROOT2, b.getLength(), DELTA);
    assertEquals(ROOT2, c.getLength(), DELTA);
    assertEquals(ROOT2, d.getLength(), DELTA);
    assertEquals(ROOT2, e.getLength(), DELTA);
    assertEquals(FLENGTH, f.getLength(), DELTA);
    assertEquals(GLENGTH, g.getLength(), DELTA);
}

@Test
public void equals() {
    assertFalse(a.equals(b));
    assertFalse(b.equals(c));
    assertTrue(a.equals(Vector.ZERO_VECTOR));
    assertFalse(f.equals(g));
}

@Test
public void vAdd() {
    Vector bc = b.vAdd(c);
    Vector cb = c.vAdd(b);

    assertTrue(bc.equals(cb));

    Vector ab = a.vAdd(b);

    assertTrue(b.equals(ab));


    assertEquals(hAddi, h.vAdd(i));
}

@Test
public void sMult() {
    float scalarA = 123.123456789f;
    float scalarB = -234.234f;
    float scalarC = 0f;
    int scalarD = 4;

    assertTrue(a.equals(b.sMult(scalarC)));

    assertTrue(new Vector(-1*scalarA, scalarA).equals(b.sMult(scalarA)));

    assertTrue(new Vector(4,4).equals(d.sMult(scalarD)));

    assertTrue(new Vector(FX*scalarB, FY*scalarB).equals(f.sMult(scalarB)));
}
```

2.3 Interfaces

2.3.1 Premise

During development of the framework, there is no way to know all possible implementations and behaviours of game objects created by an end user. We need to provide a standard blueprint that every object needs to comply with if they want to be used with the framework. This lends itself to the use of Java Interfaces. Using these, it is possible to outline a set of methods that need to be present in a game object's implementation.

2.3.2 Collidable Interface

If an object wants to be interactable with physics and collisions in the game, it must implement this interface to be compatible. The Entity class implements this by default, so some of these methods may already be implemented if the new object extends a child of Entity.

It is important to note that the getCollisionVertices() method is expected to return the vertices specified in a counter-clockwise fashion, so that if they were to be joined up first to last, the resulting shape matches your shape.

Listing 2.7: Collidable Listing

```
public interface Collidable {
    Point[] getCollisionVertices();
    BoundingBox getBoundingBox();

    boolean isSolid();
    boolean canMove();

    String getName();

    ShapeIdentifier getShapeIdentifier();

    Point getCenter();
    void setCenter(Point newCenter);
}
```

2.3.3 Renderable Interface

Whilst this framework does not include a bespoke rendering engine, the objects that come bundled with it need to be render ready so that the user can just call the relevant method for their needs.

Listing 2.8: Renderable Listing

```
public interface Renderable {
    //canvas is what the object should be drawn onto
    //renderoffset is to enable the conversion between global
    //and screen coordinates
    //secondsSinceLastRender is used for any interpolation
    //renderEntityName is a debugging flag
    void draw(Canvas canvas, Point renderOffset, float secondsSinceLastRender,
            boolean renderEntityName);

    void setColor(int color);

    void revertToDefaultColor();

    BoundingBox getBoundingBox();

    String getName();
}
```

2.3.4 Shape Interface

The framework comes with several basic shapes already implemented, but should the user wish to create new Shapes and have them be compatible with the framework's algorithms, they must implement this interface. Note: most shapes can make use of the framework's Polygon class. See section 2.11 for more detail.

Listing 2.9: Shape Listing

```
public interface Shape {
    //used for auto-recognition of parallel axes
    ShapeIdentifier getShapeIdentifier();
    Vector getForwardUnitVector();
    Point getCenter();

    //canvas is what the object should be drawn onto
    //renderoffset is to enable the conversion between global
    //and screen coordinates
    //secondsSinceLastRender is used for any interpolation
    //renderEntityName is a debugging flag
    void draw(Canvas canvas, Point renderOffset, float secondsSinceLastRender,
            boolean renderEntityName);

    Point[] getVertices();
    Point[] getVertices(Point offset);

    void setColor(int color);
    void revertToDefaultColor();

    BoundingBox getBoundingBox();

    void translate(Vector movementVector);
    void translate(Point newCenter);

    void rotate(float angle);
    void rotate(Vector v);

    int getColor();
}
```

2.3.5 InputUIElement Interface

Android uses a touch screen display, and so any custom UI elements need to keep track of the current pointer id that has been assigned to them. This is so the input manager knows which UI element needs to know about updates when the user moves their finger. Implementing this interface will provide the user with all the methods needed to define their own UI elements.

Listing 2.10: InputUIElement Listing

```
public interface InputUIElement {
    boolean containsPoint(Point eventPosition);

    void setPointerID(int id);

    boolean hasID(int id);

    int getID();

    void cancel();

    void draw(Canvas canvas, Paint textPaint);
}
```

2.3.6 IAIEntity

When the user creates an AI class, it is advisable to implement this interface also. The bundled behaviour tree nodes all reference the methods in this interface.

Listing 2.11: IAIEntity Listing

```
public interface IAIEntity {
    Weapon getWeapon();
    Vector getShootingVector();
    Point getCenter();
    Vector getForwardUnitVector();
    BoundingBox getBoundingBox();
    String getName();
    Vector getVelocity();
    void setVelocity(Vector newVelocity);
    float getMaxSpeed();
    BContext getContext();
    int getWidth();
    void setRotationVector(Vector v);
}
```

2.3.7 IMap

If the user wishes to use the bundled map package classes, then their game map class must implement this interface.

Listing 2.12: IMap Listing

```
public interface IMap {

    String getName();
    String getStage();

    int getTileSize();

    void setDoors(List<Door> doors);
    Map<String, Door> getDoors();

    void setRootMeshPolygons(List<MeshPolygon> meshPolygons);
    Map<Integer, MeshPolygon> getRootMeshPolygons();

    void setWidthInTiles(int w);
    float getWidth();

    void setHeightInTiles(int h);
    float getHeight();

    void setMeshGraph(Graph<Integer> graph);
    Graph<Integer> getMeshGraph();

    void setWalls(List<Wall> walls);
    List<Wall> getWalls();
}
```

2.3.8 Damageable

This interface needs to be implemented if the user wishes the object to take damage from certain objects in the game world.

Listing 2.13: Damageable Listing

```
public interface Damageable {

    int getHealth();

    void setHealth(int health);

    boolean inflictDamage(int damageToInflict);

    void restoreHealth(int healthToRestore);

    int getInitialHealth();
}
```

2.4 Collision Detection and Resolution

2.4.1 Introduction

A critical aspect to consider when developing a game, is the detection and resolution of collisions between objects in the game world. In this framework, I will be implementing a two stage approach to collision detection. Both of these stages will be described in detail, and will conclude with a benchmarking of the implemented algorithms.

The first stage will be to find the groups of objects that have a possibility of colliding, and ignoring the groups that have no chance of colliding. It will make use of a Spatial Partitioning (Hereby referred to as SPAT-PAT) Scheme which will place objects into the cells that the objects intersect. This stage consists of relatively inexpensive operations and I shall refer to it as the 'course grain' step.

The second stage takes the groups of objects which have a possibility of colliding (obtained from the course grain step), and performs the Separating Axis Theorem (Hereby referred to as 'the SAT') on each pair of objects within each group. This second stage will be referred to as the 'fine grain' stage from now on.

2.4.2 Spatial Partitioning

Aim & Design

The objective of this stage is to reduce the number of computationally expensive SAT operations to improve the performance of the collision detection of a game tick as a whole. A SPAT-PAT scheme will enable us to rule out pairs of objects that have no possibility of colliding, whilst highlighting objects that should be checked in the later fine grain stage.

This is achieved by dividing the game world into a series of disjoint regions, called cells which come together to form a SPAT-PAT grid. This grid is aligned with the world axes to make the insertion of objects faster as it allows the use of AABBs/Bounding Boxes (see section 1.2.1), and then placing each object into all of the cells that its AABB intersects. The size/number of the cells is not fixed, and can be changed to optimize the performance of a given game level. Note that the fewer the number of cells, the closer the resulting performance is to having no course grain step at all.

Implementation

A cell in SPAT-PAT is implemented here as a bin, and each SPAT-PAT bin has two sub-bins, one for permanently stationary objects, and one for objects that may change across ticks. For example, in Tanktrum, the objects placed in the permanent bins are walls and doors as they never move. Whereas the temporary bins will have the AI, player, bullets, and so on. The temporary bins are cleared at the start of each game tick, and filled during the course grain step of the collision detection process. The implementing code will not be shown here as it is trivial, only consisting of two sets with methods to insert and clear objects. However, the source code is accessible at the link provided at the start of this report.

The CollisionTools class contains methods to compute the insertion of both temporary and permanent objects. As these methods are quite similar, only the method for adding a temporary object will be displayed here.
The method displayed in listing 2.14 is called each time a temporary object is added to the SPAT-PAT grid. The procedure for finding which bins to add to is simply the case of iterating through each bin and doing an intersection test. As AABBs are being used, intersection tests are inexpensive.

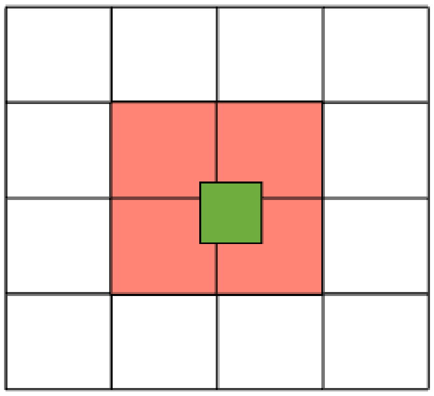


Figure 2.1: An object (in green) being placed in the SPAT-PAT grid. Bins intersecting the object are highlighted.

Listing 2.14: Insertion of an Object into Spatial Bins

```
public static boolean addTempToBins(Collidable collidable, List<SpatialBin> spatialBins){
    boolean added = false;

    for (SpatialBin bin : spatialBins){
        if (bin.getBoundingBox().intersecting(collidable.getBoundingBox())){
            //if the collidable intersects this bin's bounding box,
            //add it to the bin's temp list
            bin.addTemp(collidable);
            added = true;
        }
    }
    return added;
}
```

2.4.3 Separating Axis Theorem

Aim & Design

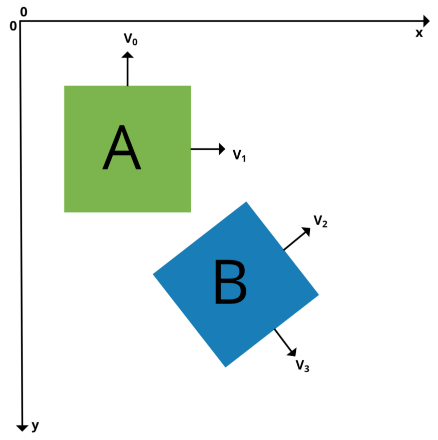
For the fine grain step of the collision detection, this framework uses the Separating Axis Theorem (SAT) [2]. The SAT says that if there exists an axis on which the projections of two objects' vertices do not overlap, then they are not intersecting. If the objects are found to be intersecting, then the algorithm returns a vector describing the smallest translation needed so that the objects are no longer intersecting.
See listing 2.15 for a formal description in pseudo-code, and figure 2.2 for a visual example.

Listing 2.15: The Separating Axis Theorem

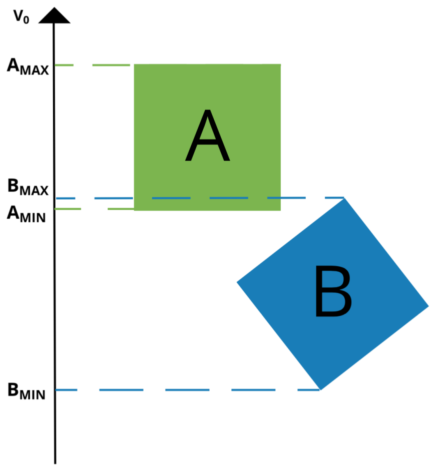
```
BEGIN SAT
    create a list for the calculated push vectors
    GET all non-parallel edges of both objects
    GET the normal vectors of each non-parallel edge
    normalize all the normal vectors

    FOR each normalized vector
        FOR each object
            project each vertex of object onto the normalized vector
            determine the MAX and MIN magnitudes of the object's projections
        END FOR

        let the first object be called A
        let the second object be called B

        IF A_MAX < B_MIN OR B_MAX < A_MIN
            this is a separating axis as the objects are not overlapping on this axis
```

```
                therefore, the objects are not intersecting.
            RETURN the zero vector
        ELSE
            this is not a separating axis
            the push vector's length is MIN of B_MAX - A_MIN AND A_MAX - B_MIN
            set the axis' vector length to be the push vector length and
                add to push vectors list
        END IF
    END FOR

    RETURN smallest vector in push vectors list
END SAT
```

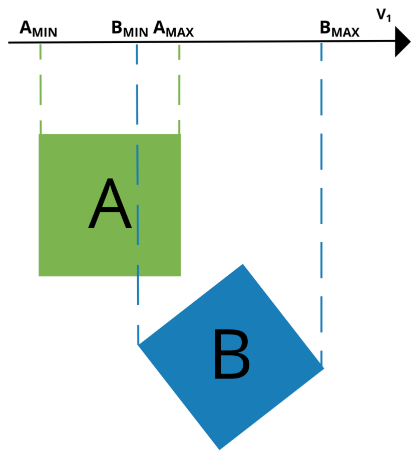
The performance of this algorithm is dependent on the number of non-parallel axes. The more axes, the longer it takes to process. However, to offset this, the early exit opportunity seen in listing 2.15 potentially allows the algorithm to save some time, however if the only separating axis is the last axis to be evaluated, then the performance doesn't change.



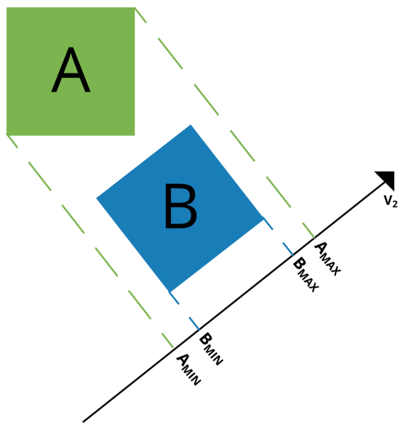
(a) Two objects, and their non parallel edge normal vectors



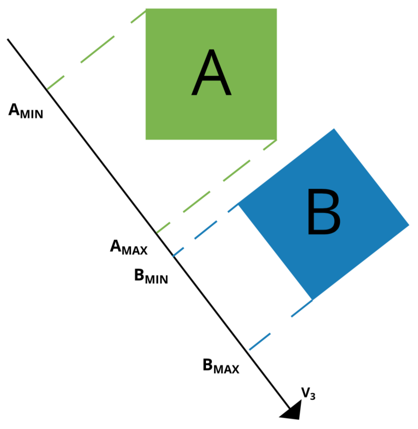
(b) Projections of A and B onto v_0, with an overlap of B_{MAX} and A_{MIN}



(c) Projections of A and B onto v_1, with an overlap of A_{MAX} and B_{MIN}



(d) Projections of A and B onto v_2, with B fully enclosed within A



(e) Projections of A and B onto v_3. No overlap between A and B and so v_3 is a separating axis

Figure 2.2: The separating Axis theorem demonstrated with two non intersecting objects. There is overlap on three out of the four axes, with the fourth being a separating axis.

A Flaw in the SAT

A drawback of this algorithm is that if an object moves sufficiently far in the space of one tick, it may jump through an object entirely, or collision resolution could place it on the wrong side of the object it collided with as shown in figure 2.3.

However, as most objects in the game world will rarely have a velocity big enough to cause this effect, it is not likely to happen and so is ignored in this implementation.

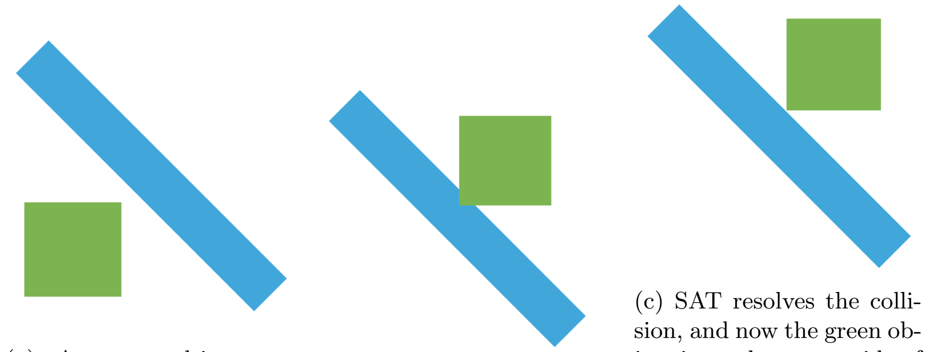

(a) A green object approaching a blue object
(b) The tick after 2.3a
(c) SAT resolves the collision, and now the green object is on the wrong side of the blue object

Figure 2.3: A flaw in the separating axis theorem

Note that this algorithm can also be used for ray casting purposes. This is possible provided that the ray is expressed as a finite line. In this case, the axes of the ray are its direction and the normal of its direction.

Implementation

The implementation of the SAT lines up with the pseudo-code outlined above (see 2.15), for the implementation see listing 2.16.

Listing 2.16: The Separating Axis Implementation

```
//checks if the max and min points on the given direction axis overlap,
//returns the overlap
public static Vector isSeparatingAxis(Vector axis, Point[] firstEntityVertices,
                Point[] secondEntityVertices){
    Pair<Float, Float> minMaxResult = projectOntoAxis(axis, firstEntityVertices);

    float firstEntityMinLength = minMaxResult.first;
    float firstEntityMaxLength = minMaxResult.second;

    minMaxResult = projectOntoAxis(axis, secondEntityVertices);
    float secondEntityMinLength = minMaxResult.first;
    float secondEntityMaxLength = minMaxResult.second;

    if (firstEntityMaxLength >= secondEntityMinLength &&
                secondEntityMaxLength >= firstEntityMinLength) {
        float pushVectorLength = Math.min((secondEntityMaxLength - firstEntityMinLength),
                (firstEntityMaxLength - secondEntityMinLength));

        //push a bit more than needed so they dont overlap in future tests
        //to compensate for float precision error
        pushVectorLength += PUSH_VECTOR_ERROR;

        return axis.getUnitVector().sMult(pushVectorLength);
    }
    return Vector.ZERO_VECTOR;
}

public static Pair<Float, Float> projectOntoAxis(Vector axis, Point... vertices){
    float minLength = Float.POSITIVE_INFINITY;
    float maxLength = Float.NEGATIVE_INFINITY;

    float projection;
    for (Vector vertexVector : convertToVectorsFromOrigin(vertices)){
        //iterate through the vertices, projecting them onto the axis,
        //keeping the maximum and minimum projections as you go

        projection = vertexVector.dot(axis);

        minLength = Math.min(minLength, projection);
        maxLength = Math.max(maxLength, projection);
    }

    return new Pair<>(minLength, maxLength);
}
```

2.4.4 Discrete versus Continuous

There are two classifications of collision detection approaches: discrete and continuous. Both SPAT-PAT and the SAT can be applied in the context of either classification with a few adjustments.

Discrete collision detection evaluates the game objects at discrete time steps, usually the time of a game tick. It doesn't take the future of the object into account, just it's current position. For this reason, discrete approaches are usually faster as they do not need to interpolate between two positions. However, if an object is moving sufficiently fast, it may pass over another object in between game ticks meaning the collision won't be detected.

Continuous collision detection takes the predicted future position of the object in the next game tick based off of it's current velocity, and uses both positions when evaluating collisions. This method is used for projectiles, for example.

2.4.5 Projectile Collision Detection

Design

As outlined in section 2.4.4, sometimes collision detection needs to be continuous. The only objects currently employing this method are objects extending the Projectile class. As a projectile does not need to bounce, all it needs to do is be destroyed when it hits something, the following method suffices.

This method encases the two positions in a bounding box, so that any objects that lie in-between the positions, are detected as collisions.
Start by joining the two circles with a vector u, the magnitude of u is the displacement of the projectile. Using the center of u, an orientated bounding box can be constructed with width equal to a diameter of the projectile, and length equal to the magnitude of u summed with a diameter of the projectile. (See figure 2.4)

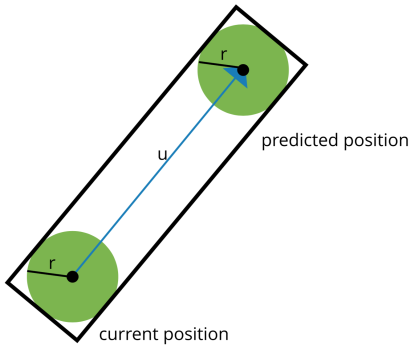


Figure 2.4: Projectile Prediction Bounding Box

Implementation

The implementation of this algorithm can be seen in figure 2.17.

Listing 2.17: The method for forming a projectile's orientated bounding box

```
public Point[] getCollisionVertices() {
    float width = getRadius() * 2;
    Vector v = new Vector (currentCenter, nextTickCenter);
```

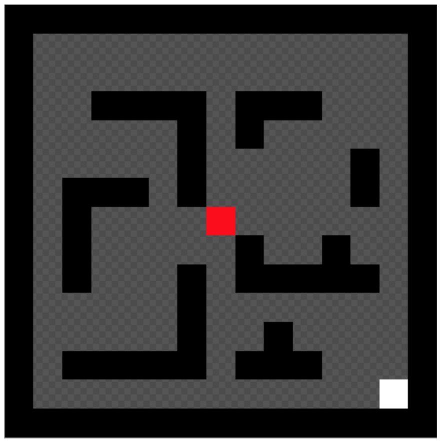


Figure 2.5: The game map used in the benchmarking tests. Black tiles are walls, the red tile is the AI spawn point, and the white tile is the player spawn point

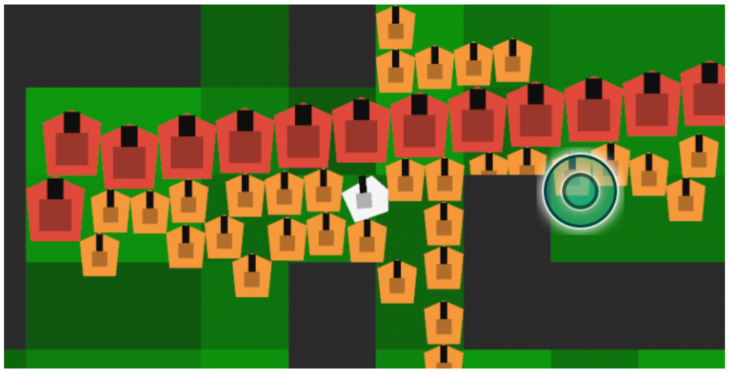


Figure 2.6: Existing AI being shoved to accommodate new AI spawning in on the same spawn point

```
    float length = v.getLength() + getRadius() * 2;

    Point center = v.sMult(0.5f).getHead();

    r = new Rectangle(center, width, length, Color.BLACK);

    r.rotate(v.getUnitVector());

    return r.getVertices();
}
```

2.4.6 Benchmarking

Introduction

The following section will provide a performance analysis of the two stage collision detection outlined in the previous sections. The tests were conducted in Tanktrum on one map that is used as the control.

The control map (see figure 2.5) consists of 20 wall entities (See section 2.8 for details on how walls are formed), 50 AI entities, and the player, bringing the total number of entities on the map to 71. For the duration of these tests, the AI processing has been disabled as to not interfere with the results.
A time entry in table 2.1 is calculated by logging the time each collision stage took in each game tick for 5 seconds, and then taking the average. Each benchmark was taken multiple times until they produced a consistent result, to allow the entities to settle as they spawn due to them all sharing the same spawn point as seen in figure 2.6. In the below benchmarks, the number of SPAT-PAT bins is increased and the time for the collision tick to execute is logged.

The times obtained in table 2.1 were then plotted as graphs and can be seen in figures 2.7 and 2.8. As can be seen from the graph, having more cells/bins makes the execution time of the fine grain step decrease significantly. It can also be seen

Table 2.1: Table showing the change in collision check times as SPAT-PAT grid cell count increases

Cell Count	Course Grain Time (ms)	Fine Grain Time (ms)	Total Time (ms)
1	0	54	54
4	0	30	30
9	1	25	26
16	1	22	23
25	2	16	18
36	3	14	17
49	4	13	17
64	6	10	16
81	7	10	17
100	8	9	17
121	11	8	19
144	11	10	21

Table 2.2: Table showing the change in the number of collision checks as SPAT-PAT grid cell count increases

Cell Count	Course Grain Checks	Fine Grain Checks
1	51	2295
4	204	1121
9	459	968
16	816	782
25	1275	588
36	1836	557
49	2499	522
64	3266	379
81	4136	369
100	5100	284
121	6184	307
144	6178	314

that there exists an optimal number of bins to have in the level as the graph flattens out. This optimal value will change based on the number of objects in the level, and the size of the level itself.

By looking at table 2.2 it is evident how expensive a fine grain check is compared to a rough grain check. Consider also that the time to perform a rough grain check is the same for an object regardless of the number of sides the object has, as the check is performed with the object's AABB which is always a rectangle. Whereas the fine grain check time increases with the number of non-parallel sides on an object.

Note that having only one SPAT-PAT bin is equivalent to having no course grain step at all. Only adding the temporary objects each step is also evident in table 2.2 as when there is only one cell, 51 checks are made which is equal to the number of AI and the player.

2.4.7 Concluding Thoughts

The algorithms in use provide a versatile and reliable method for detecting collisions. A game that wishes to make use of these methods has to make it's objects comply with the Collidable interface, then call the relevant collision method when applicable. This system functions efficiently, as demonstrated with the benchmarks and I am overall satisfied with the implementations.

However, it is not without faults. The separating axis theorem has a flaw as outlined above where it does not know which side a colliding object came from, it just resolves the collision with the smallest vector possible. In future I will expand the collision detection step to also find the point of intersection, by using continuous approaches.

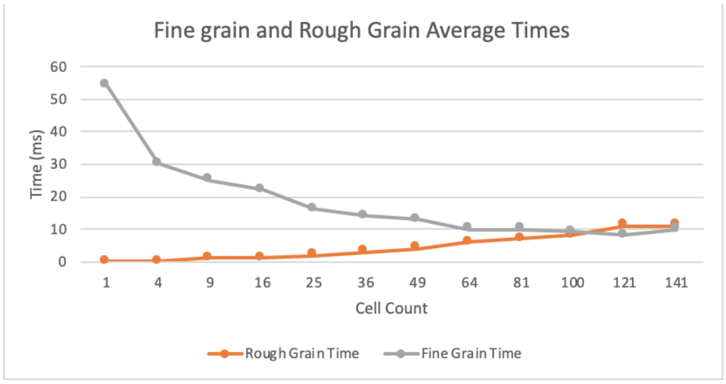


Figure 2.7: Graph of SPAT-PAT Cell Count Against Fine Grain and Course Grain Times

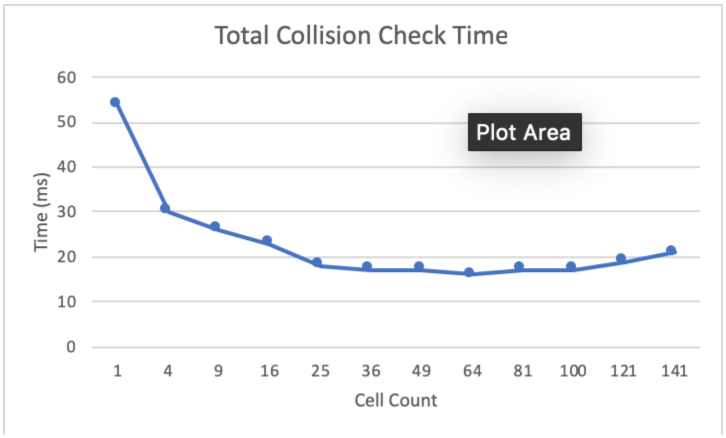


Figure 2.8: Graph of SPAT-PAT Cell Count Against Time

2.5 AI Navigation Mesh

2.5.1 Introduction

Being able to navigate through the game world is a critical aspect of a game AI, and it presents the challenge of AI travelling from point A to point B without colliding with other game entities, or getting stuck in a dead end. The algorithms for plotting a path for AI to follow is outlined in section 2.6 however, these algorithms rely on a graph as input, and the problem of how the AI will avoid obstacles whilst following a path remains unsolved.

If movement was tile based (Example: Figure 2.9), a basic approach would be to take the grid of tiles containing all objects in the game world and use it as a graph for the path-finding algorithm. However if the framework only provided support for tile based games, the potential applications would be limited since most games released these days allow some form of continuous movement. This means that the framework needs to support continuous movement as well as tile based.

Continuous movement does not have an immediately obvious system in place to use as an input for path-finding algorithms. One approach would be to employ a spatial partitioning scheme like in section 2.4.2, with cell size roughly equal to the size of an average game object. This would provide a grid to apply path-finding on, however as game map sizes grow larger, the number of nodes in this grid will increase exponentially.

It is important to note that whilst a spatial partitioning scheme is inexpensive when cells are at the scale used in collision detection, in this context the cells are significantly smaller meaning it can still perform poorly. To combat this inefficiency problem on large levels, a data structure called a navigation mesh (navmesh) [3],[4] was designed. The aim of the navmesh is to provide a set of polygons in which the AI is free to roam without needing to detect collisions with other game objects. The mesh consists of a set of disjoint polygons, which are connected via edges to any neighbouring polygons. This set of disjoint polygons and connections can be used as a graph for path-finding algorithms.
The navmesh is created manually by the developer or generated based on the level design and a set of constraints. Due to the increased workload if the navmesh is created manually, I elected to create an algorithm to generate the navmesh automatically.

Usually a navmesh is used in 3D environments [4] and because of this, is usually generated using triangulation techniques [5],[6] to account for changes in the ground elevation. As this framework is only for 2D games, using triangulation can potentially over-complicate the issue as we do not need to take depth into account. It is for this reason that I elected to design my algorithm to calculate a set of disjoint AABBs to serve as my polygons. As the polygons are axis-aligned, insertion

Figure 2.9: Example of a tile based movement system. Credit: https://www.jonathanyu.xyz/2017/01/14/data-structures-for-tile-based-games/

of game objects into the grid is an inexpensive AABB intersection test.

The following algorithm generates a navmesh from an inputted grid representation of the level containing information about what cells are blocked by static impassable objects. This does not mean that the resulting level has to be tile based, the mesh that is generated will function on a continuous plane as seen in Tanktrum.

2.5.2 Design

The aim of this algorithm is to produce a set of disjoint axis-aligned rectangles which are joined to adjacent rectangles forming a graph. A requirement is that an AI entity should be able to move from any point in one polygon to any point in a neighbouring polygon, without colliding with any other entities in the level.

The algorithm consists of three main steps: a vertical scan, a horizontal scan, and calculating the intersection of the two scans.
For a description of a vertical scan see listing 2.18. A horizontal scan is conceptually the exact same as a vertical scan except:

- Rows are scanned instead of columns
- Columns are walked across instead of rows
- x coordinates are evaluated in the rules instead of y coordinates

After conducting a vertical scan and a horizontal scan, we end up with two separate sets of polygons. One set has polygons sensitive to changes in vertical distances to blocked cells, and the other is sensitive to changes in horizontal distances to blocked cells.
In order to construct a navmesh that is sensitive to changes in both vertical and horizontal distances to blocked cells, we take the intersection of both generated sets. This can be seen in figure 2.10.
After applying this algorithm, we end up with a navmesh that fulfills the requirements we set out to complete.

Listing 2.18: Vertical Scan Pseudo-code

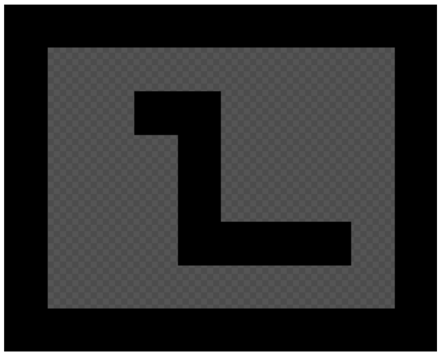
```
FOR each column of the inputted grid
    Find an unblocked cell, if there are none, move to the next column
    Find the current unblocked section by walking through the rows of the column,
    until a blocked cell is encountered, counting the number of cells traversed.

    IF there are any polygons from the last column scanned
        check each of them for if they comply with the following rules:
            They start on the same y coordinate as the current section
            They end on the same y coordinate as the current section

        IF a complying polygon is found
            add the current section to that polygon,
        ELSE
            create a new polygon for the section
        ENDIF
    ELSE
```

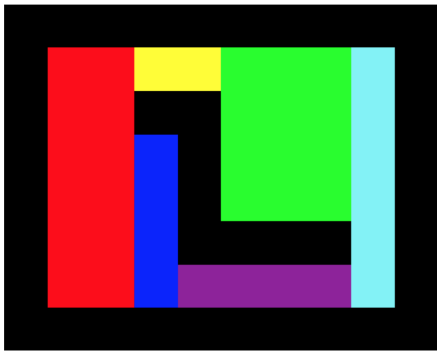
```
        add the current section to a new polygon
    ENDIF

    REPEAT the above if there are any unblocked cells after the
    end of the current section
ENDFOR
```

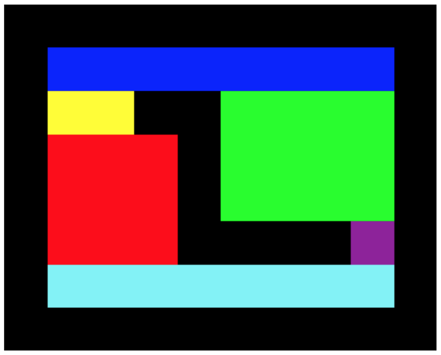
At any step in a scan, a section can be in the same polygon as a section in the previous scan line if they comply with the following rules. The coordinate referenced is dependent on the scan taking place. If it is a horizontal scan then the x coordinates are evaluated, similarly vertical scans evaluate y coordinates:

- They start on the same coordinate.
- They end on the same coordinate.

A section is defined as a continuous strip of unblocked cells from the starting cell in a row/column to the next blocked cell.


(a) Map without mesh overlay


(b) Polygons generated after vertical scan


(c) Polygons generated after horizontal scan

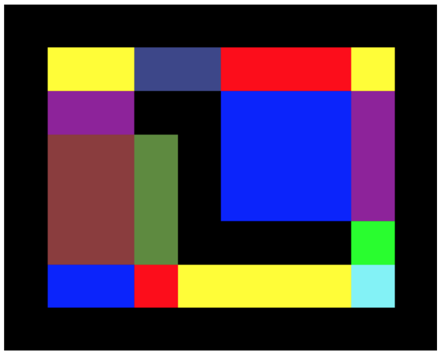

(d) Complete mesh after taking intersection of polygons generated in 2.10b and 2.10c.

Figure 2.10: Navigation Mesh Generation Applied to a 10x8 Map

2.5.3 Implementation

MeshSet Class

This class is only used during the execution of the navmesh generation. The notable sections can be seen in listing 2.19. Both vertical and horizontal scans use this class to store information about the generated polygons.
A list of the contained cells is maintained, with helper methods for adding new cells to this list. The tile which was initially added when the object was created is also stored along with the distance till the next blocked cell. This allows fast access to the information needed when evaluating whether to add a cell to this object or not.

Listing 2.19: MeshSet class

```
private int id;
private List<Tile> containedTiles;
private Tile startTile;
private int distanceToWall;

...

public List<Tile> intersection(MeshSet b){
    List<Tile> intersectionSet = new ArrayList<>();

    for (Tile tile : containedTiles){
        if (b.containedTiles.contains(tile))
            intersectionSet.add(tile);
    }
    return intersectionSet;
}
```

MeshPolygon Class

This class is used after the generation algorithm has completed. It is the object that is stored in the game level state and used for intersection tests with game objects when accessing the mesh. The notable sections of this class can be seen in listing 2.20.
The object is instantiated using an instance of MeshSet and an argument for scaling the tile objects to global coordinates. The bounding volume for the polygon is calculated by sorting the contained tiles and calculating the maximum and minimum points for both x and y axes. The id field is used in the navmesh graph and to enable the object to be stored in a Map for efficient access.

Listing 2.20: MeshPolygon Class

```
private int id;
private Rectangle area;
private Random random;

public MeshPolygon(MeshSet meshSet, int tileSize){
    this.id = meshSet.getId();

    this.random = new Random();

    generateBoundingVolume(meshSet.getContainedTiles(), tileSize);
}

private void generateBoundingVolume(List<Tile> meshSetTiles, int tileSize){
    //sort tiles primarily by y coordinate, if equal the x coordinate is then compared
    meshSetTiles.sort((a,b) -> {
        if (a.getY() < b.getY())
            return -1;
        else if (a.getY() > b.getY())
            return 1;
        else{
            return Integer.compare(a.getX(), b.getX());
        }
    });

    //the stored tiles for top left and bottom right respectively
    Tile tl = meshSetTiles.get(0);
    Tile br = meshSetTiles.get(meshSetTiles.size()-1).add(1,1);

    //the tiles above converted to global coordinates
    Point topLeft = new Point(tl).sMult(tileSize);
    Point bottomRight = new Point(br).sMult(tileSize);

    Point center = topLeft.add((bottomRight.getX() - topLeft.getX())/2f,
            (bottomRight.getY() - topLeft.getY())/2f);

    this.boundingVolume = new Rectangle(center, topLeft, bottomRight,
    ColorScheme.randomGreen());
}
```

Generation Algorithm

The inputted grid is implemented as a 2D List of integers. The value of the integer determines how the algorithm will treat that cell.

- If it is -1: Considered to be a blocked cell
- If it is 0: Considered to be an empty and unblocked cell
- If it is -2: Considered to be a door and an unblocked cell

Doors are considered unblocked so that the generated graph can be modified when a door is locked or unlocked. If a door is locked, the edges leading into the polygon the door lies in are removed meaning no AI entities will plot a path through the door, but any AI in the doorway will know how to navigate out from it.
Doors are a special case, they cannot move but can either be passable or not, depending on if they are locked. For this reason, a door must lie in it's own polygon. To ensure this, a postprocessing step occurs in the constructMesh method. It can be seen in listing 2.21.

Listing 2.21: constructMesh method that is called to execute generation of the navigation mesh

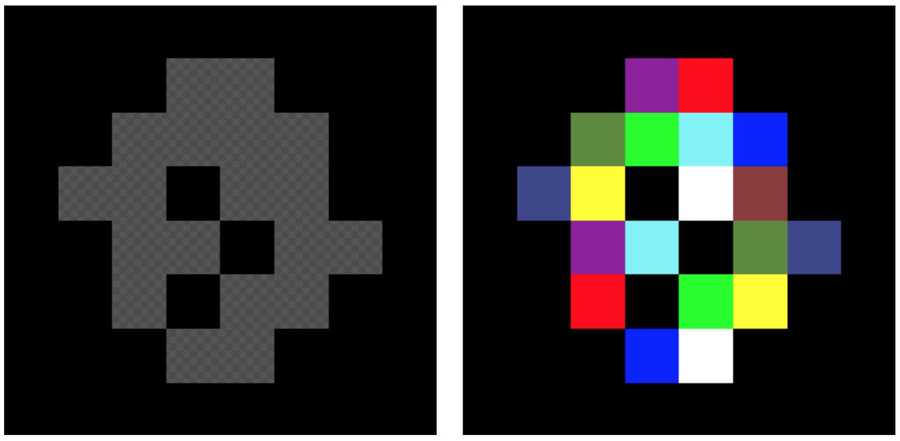
```
public void constructMesh(List<List<Integer>> tileList, List<DoorBuilder> doorBuilders,
                int tileSize){
    //tileList is the grid of blocked, unblocked cells
    //doorBuilders contains the cells of the doors in the level
    //tileSize is used when converting the navmesh polygons to world coordinates


    this.tileList = tileList;
    this.tileSize = tileSize;

    //do a horizontal scan
    currentScan = Scan.HORIZONTAL;
    hScan();

    //do a vertical scan
    currentScan = Scan.VERTICAL;
    vScan();

    //intersect the scans
    intersectScans(hScan, vScan);

    //postprocessing step
    //make sure doors are in their own set by iterating through the door spawn cells
    for (DoorBuilder doorBuilder : doorBuilders){
        Tile doorTile = doorBuilder.getLiesOn();

        MeshSet doorSet = null;

        //find the meshset that the door is in
        for (MeshSet meshSet : meshIntersectionSets){
            List<Tile> containedTiles = meshSet.getContainedTiles();

            //if the meshset has tiles other than the door tile
            if (containedTiles.contains(doorTile) && containedTiles.size() > 1){
                //remove the door from the set
                meshSet.removeTile(doorTile);
                //create a new set for the door
                doorSet = new MeshSet(intersectionId.id(), doorTile, 0);
            }
        }

        if (doorSet != null){
            //if the door was added to a new set, add this new set to the graph as a node,
            //and to the intersection sets
            meshIntersectionSets.add(doorSet);
            meshGraph.addNode(doorSet.getId());
        }
    }

    constructMeshPolygons();
    constructGraph();
}
```

Listing 2.22: Implementation of the Horizontal Scan

```
private void hScan(){

    List<MeshSet> previousMeshSets = new ArrayList<>();

    for (int rowIdx = 0; rowIdx < tileList.size(); rowIdx++){

        List<Integer> row = tileList.get(rowIdx);
        List<MeshSet> currentMeshSets = new ArrayList<>();

        Tile startingTile = findOpenCellOnRow(rowIdx, 0);

        //whilst the row has no unchecked starting tiles
        while (startingTile != null) {
            //the row must have a non blocked cell

            //the distance to the next blocked cell after the startingTile in this row
            int distanceToWall = getHDistanceToWall(startingTile.getX(), row);

            //the meshset currently being added to
            MeshSet activeMeshSet = getActiveMeshSet(startingTile, distanceToWall,
                    previousMeshSets);
            currentMeshSets.add(activeMeshSet);

            //at this point, we have the meshset to add cells to, so walk along the row
            //from the starting cell (inclusive) until a wall is hit.

            //the column index that the scan loop exited at,
            //so we can resume the loop right after the current section
            ExitValue loopExitValue = new ExitValue(row.size()-1);

            int currentCell;
            for (int columnIdx = startingTile.getX(); columnIdx < row.size(); columnIdx++)
            {
                currentCell = row.get(columnIdx);
                //loopExitColumnValue = columnIdx;
                loopExitValue.setExitVal(columnIdx);

                if (analyseCell(currentCell, new Tile(columnIdx, rowIdx), activeMeshSet,
                        loopExitValue))
                    break;
            }

            //if the exit value of the scan loop didnt reach the end of the row,
            //then find the next starting tile
            //if it did, then the row is finished,
            //setting startingTile to null will end the loop for this row.
            if (loopExitValue.getExitVal() < row.size()-1)
                startingTile = findOpenCellOnRow(rowIdx, loopExitValue.getExitVal());
            else
                startingTile = null;

        }

        //iteration is finished, moving onto next row. save state of this row
        previousMeshSets = currentMeshSets;
    }
}
```

2.5.4 Performance

One aim of this algorithm is to reduce the number of nodes that a path-finding algorithm has to visit in order to plot a path. With this algorithm it is clear to see that in the worst case, the generated mesh is equal in performance to having a spatial partitioning scheme in place, as the individual polygons will only be as large as one cell in the inputted grid (See figure 2.11). The cause of this drop in performance comes from the many changes in the distances between walls. The smoother a map's walls are, the less polygons there are, meaning better performance.



(a) Empty Map (b) Generated Polygons for mesh

Figure 2.11: Worst case scenario of navmesh on an 8x8 Map

This means that almost always, there will be a positive performance boost on path-finding, especially on larger maps (See figure 2.12).

See table 2.3 for a view on how many polygons there are to traverse compared to individual cells. The difference isn't much in terms of raw numbers for smaller maps, but of larger maps the difference is more evident.

Table 2.3: Table showing the effects of navmesh generation on different maps

Figure Reference of map	Empty Cells	Generated Polygons	Polygons/Cells
2.10	40	14	0.35
2.12	596	180	0.3
2.11	19	19	1

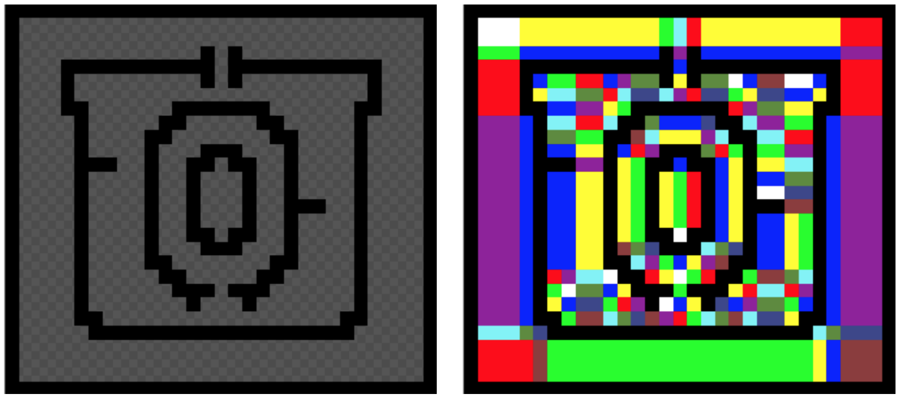

(a) Empty Map (b) Generated Polygons for mesh

Figure 2.12: Generation Algorithm Applied to a 31x28 Map

2.5.5 Afterthoughts

This algorithm performs as designed, and is guaranteed to produce a navmesh allowing the AI to move through the map unhindered. It is worth noting that the only objects taken into account for this process are ones provided at level launch, so cannot include movable or temporary objects such as the player, other AI, bullets, and so on.
This is not as big of a shortcoming as it may first appear as these listed entities will rarely ever affect how the AI plots a path through the map, since steering behaviours [7] can be employed to navigate round short range small obstacles. The navmesh allows the AI to be unaware of the majority of game objects which are namely walls and doors in the base implementation of Tanktrum.
Moving forward, to increase the flexibility of the navmesh I would implement a sub-partitioning scheme for the polygons so that entities such as the player or other AI can be more effectively avoided. The reason this was not implemented at the time of writing this report is due to time limitations, and also due to the added time cost of re-evaluating the plotted path each tick to make sure that if any entities have moved, the AI is not now on a course to collide with them. Using the current version, the AI only has to plot a path once, avoiding overusing expensive operations.

2.6 AI Path-finding

2.6.1 Introduction

For game AI to navigate through an environment without getting stuck on other game objects or navigating into hazards, we need to have a method for avoiding such obstacles. Whilst there exist methods for navigating through the game world whilst avoiding obstacles and other objects [7], they are not often the only navigation tool used by AI. The main drawback of such approaches, is that they do not have a planning aspect to them. They are simply reactionary devices for any encountered obstacles. This is not to say that they are without purpose as some of these methods are used in this framework as a secondary device. However we need a more sophisticated approach to navigation and that comes in the form of path-finding.

Path-finding algorithms use a graph (or grid treated as a graph) to evaluate the game world and plot a route through for the AI to follow.
There are many different approaches, the commonly used ones are outlined in section 2.6.3, but for this framework I will be implementing the most popular, and the most relevant one which is called A* (see section 2.6.4).

The A* algorithm will be fed the graph from the framework's mesh generation algorithm, and so will change direction often, and at irregular angles which doesn't look like natural movement when the AI follows it. This lead me to develop a post processing step using Bézier curves in order to smooth out the path the AI will follow.

2.6.2 Graph classes

I have implemented a directed graph data structure for use with the chosen path-finding algorithm. With it being directed, I am able to remove edges going into a node, whilst leaving the edges to travel from a node. This allows me to stop AI from entering a certain zone, but allow AI already in the zone to leave.

My graph implementation is a generic set of classes so that I can supply any object for use as the Node id. The A* algorithm I have implemented is generic, however when used for the navmesh, it is of type Graph¡Integer¿ where the node id is the id of the corresponding polygon in the navmesh.

I chose to make the implementation generic so that should the user require a graph package, they don't need to import additional modules. The fact that the node id is generic means that you can make the graph store anything, for example a graph of Points or Tiles.

Graph class

The Graph class contains helper methods for adding and removing nodes and edges, along with the A* algorithm itself, see listing 2.23 for the constructor and fields.

Edge class

Since the implementation is of a directed graph, and edge consists of a from node and a to node. The edge can only be traversed in one way, see listing 2.23 for the constructor and fields.

Node class

A node consists of a node id and a list of connections, see listing 2.23 for the constructor and fields.

Listing 2.23: Constructors and fields for the graph package

```
public class Graph <T> {
    private Map<T, Node<T>> nodes;
    private UniqueID uniqueID;

    public Graph(){
        this.nodes = new HashMap<>();
        this.uniqueID = new UniqueID();
    }
}

public class Node<T> {
    T nodeID;

    List<Edge<T>> connections;

    public Node(T nodeID){
        this.connections = new ArrayList<>();

        this.nodeID = nodeID;
    }
}

public class Edge <T>{
    private int edgeId;
    private Node<T> fromNode;
    private Node<T> toNode;

    private int weight;

    public Edge(int edgeId, Node<T> fromNode, Node<T> toNode, int weight){
        this.edgeId = edgeId;

        this.fromNode = fromNode;
        this.toNode = toNode;

        this.weight = weight;
    }
}
```

2.6.3 Commonly Used Path-finding Algorithms

Breadth First Search

Breadth first search is one of the simplest graph search algorithms. It branches out from the start node visiting the nodes with the smallest distance from the start first. This algorithm doesn't take the edge weights of a graph into account, and as it has to visit every node, it is very inefficient on larger graphs. A way to improve performance would be to include an early exit if it finds the target, but as it doesn't take graph costs into account, it may not be the shortest or most optimal.

Greedy Best First Search

Greedy best first search operates with the goal of trying to get as close to the goal as possible, using a heuristic function to estimate the distance to the goal from a particular node. There are many functions you can use as a heuristic, but the most common are the Manhatten distance (if using a grid with 4 way travel) and Euclidean distance (using the Pythagorean Theorem).

This algorithm always heads down the lowest cost path, and because of this can get stuck in dead ends and have to backtrack out.

Dijkstra's Algorithm

Dijkstra's algorithm focuses on the cost to get to a particular node from the start node. For each cell it visits, it keeps track of the node that it came from, and the cost it took to get there. This cost is calculated by summing the edge costs traversed along the path it took.

This approach means that it is biased towards taking the lower cost paths, meaning that you can assign higher costs to nodes that you want the algorithm to avoid. Dijkstra's is a powerful algorithm, but the fact it isn't actively trying to get close to the target means it can end up visiting more nodes than necessary.

2.6.4 A* Algorithm

Introduction

The A* algorithm [8],[9] is the path-finding method I have chosen to implement for this framework. I chose it because it has qualities of both Greedy best first search and Dijkstra's algorithm. It considers the distance travelled so far (like Dijkstra's), and also uses a heuristic function to estimate how far the target is from the current node (like Greedy Best First Search).

Implementation

Whilst there isn't much to change about the actual algorithm, I have made my implementation for a generic class, and so it can be applied to any graph made using this data structure. The heuristic function is passed in as an argument to enable the user to have a large degree of control over how the A* is executed as the heuristic can change the plotted path produced by the algorithm. The IHeuristic is a lambda interface consisting of one method that takes two nodes of type T and returns an int.

```
public List<T> applyAStar(Node<T> startNode, Node<T> targetNode,
            boolean returnIncompletePath, IHeuristic<T> heuristic){

    //creates a priority queue based on a nodes's fCost (Lowest cost at head)
    PriorityQueue<Pair<Node<T>, Integer>> openSet;

    //each key is the tile coordinate of a tile. It's value is the gcost spent to
    //get to that tile from the start

    //stores the cost to get to a particular node from the start node
    Map<Node<T>, Integer> costSoFar;

    //each key is a child node, it's value is it's parent node,
    //i.e. the node that comes before the key node in the current path
    Map<Node<T>, Node<T>> cameFrom;

    openSet = new PriorityQueue<>(10, (a, b) -> {
        if (a.second < b.second)
            return -1;
        else if (a.second.equals(b.second))
            return 0;
        return 1; });
    cameFrom = new HashMap<>();
    costSoFar = new HashMap<>();
```

```
//initialise sets by adding the start node with 0 cost
openSet.add(new Pair<>(startNode, 0));
costSoFar.put(startNode, 0);

//is given a value of null so when backtracking the start point
//can be identified
cameFrom.put(startNode, null);

if (startNode.equals(targetNode)){
    //target
    //forms a list from the found path
    return tracePath(formPathStack(cameFrom, targetNode));
}

while (!openSet.isEmpty()){
    //while there are nodes to be checked


    Pair<Node<T>, Integer> currentPair = openSet.poll();
    Node<T> currentNode = currentPair.first;
    int currentCost = currentPair.second;

    for (Edge<T> connection : currentNode.getConnections()){
        //iterate through the current node's neighbours

        Node<T> neighbour = connection.traverse();

        if (targetNode.equals(neighbour)){
            //if the next node is the target, add it to the cameFrom
            //map and return the path generated
            cameFrom.put(neighbour, currentNode);
            return tracePath(formPathStack(cameFrom, targetNode));
        }

        //calculate the cost to get to this neighbour from the start node
        int gCostToNext = currentCost + connection.getWeight();

        //If the tile hasn't been visited before, or the cost to get to this
        //tile is cheaper than the already stored cost
        //add it to all tracking sets
        if (!costSoFar.containsKey(neighbour) || costSoFar.get(neighbour) >
                    gCostToNext) {

            //calculate the total cost to get to this node
            //edge costs + heuristic
            int fCost = gCostToNext + heuristic.calc(currentNode.getNodeID(),
                    neighbour.getNodeID());

            //update the cost so far for this node for the current path
            costSoFar.put(neighbour, gCostToNext);
            //add this neighbour to the openset to be evaluated
            openSet.add(new Pair<>(neighbour, fCost));
            //update the current path with the neighbour and it's parent
            //which is the currentNode
            cameFrom.put(neighbour, currentNode);
        }
    }
}

//no path could be found, return an empty list
```

```
    return new ArrayList<>();
}
```

2.6.5 Bézier Interpolation

Introduction

The use of the A* algorithm results in a usable path that an AI can follow. However, as can be seen in figure 2.13, the resulting path is jagged and would look unnatural if the AI were to follow it. This lead me to develop a post processing step for the path-finding algorithm.

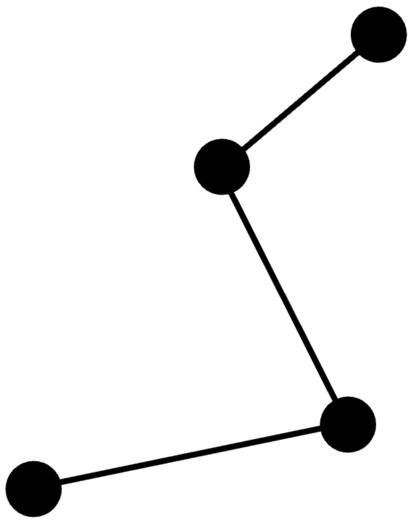


Figure 2.13: Potential path shape provided by the A* algorithm, notice the shape turns after a node.

This post processing step will smooth out the path, and make the resulting AI movement flow more naturally.

Design

The algorithm outlined in this section makes use of quadratic Bézier curves. A Bézier curve is a parametric way of expressing a curve. It is defined by three 'control' points, which dictate the form of the resulting curve.

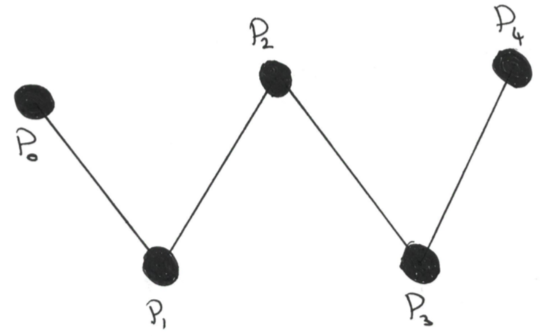
By choosing a discrete change in the parameter t, we can walk along the curve and gain an interpolated value for each value of t.

The basic concept of the algorithm is to use three successive nodes as control points in a Bézier curve, and interpolating across the curve at some chosen Δt, until t is equal to 1, then moving two nodes forward in the path, and repeating.

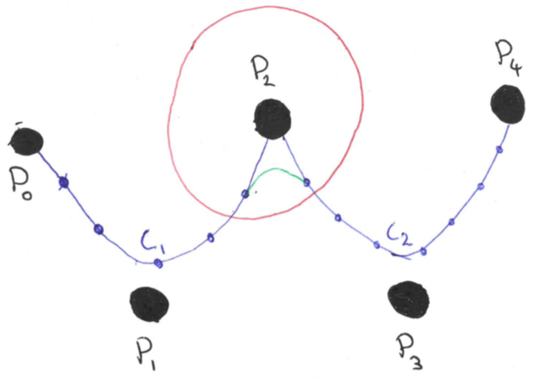
Whilst this will smooth out a curve partially, at the node where two curves meet there will still be a large change in direction which is what this algorithm is trying to remove. To overcome this issue, we form a secondary curve which starts at the last interpolated point of the primary curve, and ends on the first interpolated point of the next curve, using the last node in the primary curve as P_1.

$$B_3(t) = (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2 \tag{2.1}$$

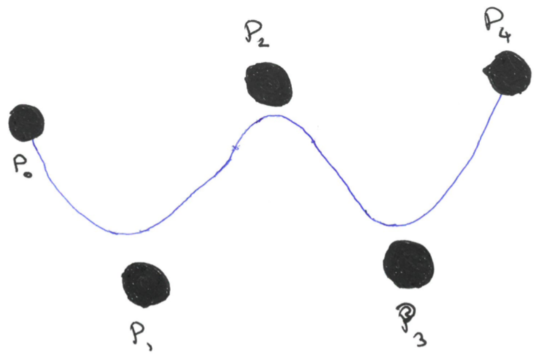
Figure 2.14: Quadratic Bézier curve $B_3(t), t \in [0,1]$ given by three control points P_0, P_1, P_2



(a) Path calculated by the A* algorithm



(b) Primary Bézier Curves are in blue, secondary curve is in green. Red circle is highlighting abrupt change in direction using only primary curves.



(c) Result of applying the interpolation algorithm

Figure 2.15: The Path Interpolation Process

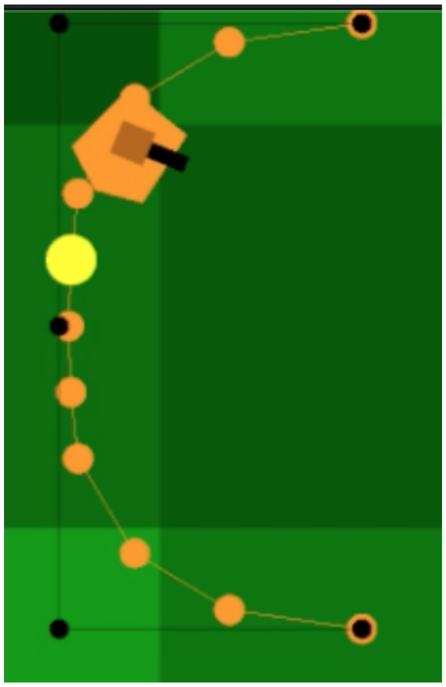


Figure 2.16: Example of interpolation algorithm in Tanktrum. Highlighted in orange are the interpolated points, in black are the original A* path points, the large yellow point is the point currently being targeted.

Implementation

The implementation of this algorithm can be found in listing 2.24, along with comments outlining the process being carried out.

Listing 2.24: Implementation of Bézier interpolation algorithm

```
private void interpolatePath(){
    if (basePath.size() >= 3){
        //there are enough nodes in the path to perform interpolation at least once.

        //stores the last interpolated value
        Point lastIPoint = null;

        //add the starting node
        interpolatedPath.add(basePath.get(0));


        for (int i = 0; i < basePath.size() - 2; i+=2){
            //visit every other node in the basePath

            //the starting value of t
            float t = 0.25f;

            //create a new Bezier curve using the current path point,
            //and the following two path points
            QCurve qCurve = new QCurve(basePath.get(i), basePath.get(i+1),
                    basePath.get(i+2));
            //evaluate the curve at the starting value of t, but do not add it to
            //the interpolated path
            Point p = qCurve.evalQCurve(t);


            if (lastIPoint != null){
                //if this is not the first curve to be evaluated
                //form a new curve starting at the last interpolated point of the
                //previous curve,
                //ending at the first interpolated point of qCurve.
                QCurve qCurve1 = new QCurve(lastIPoint, basePath.get(i), p);

                interpolateCurve(qCurve1);
            }

            //evaluate qCurve at the fixed time step
            lastIPoint = interpolateCurve(qCurve);
        }

        if (basePath.size() % 2 == 0){
            //if the path is an even length, the last node wont have been interpolated
            QCurve qCurve = new QCurve(lastIPoint, basePath.get(basePath.size()-2),
                    basePath.get(basePath.size()-1));
            interpolateCurve(qCurve);
        }

        //add the last node to the path, as the value of t never reaches 1.
        interpolatedPath.add(basePath.get(basePath.size()-1));
    } else {
        //there aren't enough points to interpolate
        interpolatedPath = basePath;
    }
}

private Point interpolateCurve(QCurve qCurve){
    float t = 0.25f;
    Point lastIPoint = null;

    while (Float.compare(t, 1f) < 1){
        //evaluate the curve whilst t is less than 1

        Point p = qCurve.evalQCurve(t);

        if (Float.compare(t, 1f) != 0) {
            //add the interpolated point to the path provided the value of t
            //is not equal to 1
            interpolatedPath.add(p);

            //update the last interpolated point field
            lastIPoint = p;
        }

        t += 0.25;
    }

    return lastIPoint;
}
```

Afterthoughts

The interpolation algorithm has been a successful implementation, as it works as intended, smoothing out the A* plotted paths. However, one key change that will be made going forward will be the ability to dynamically change the interpolation step size depending on the need of the end user.

2.7 AI Behaviour Trees

2.7.1 Introduction

At this point in the report, the user has tools to construct an environment that the AI is safe to navigate around in, and an algorithm for plotting-paths for their AI to follow. However these tools do not function as a brain for the AI, simply a way for them to interact with the world around them. This brings up the problem of implementing behaviour for the AI to follow. For example, plotting a path to the player and shooting them on sight, or fleeing to an ally for help if their health gets below a certain threshold.

The simplest approach to this problem would be to have the behaviour contained in the AI class as a block of conditional statements. The problem with this approach however is that large behaviours can easily contain a lot of if statements, leading to large and unsightly code that is difficult to maintain. It also doesn't follow good code practices as the code is not easily re-used in another AI.

A more sophisticated approach is that of the behaviour tree [10]. Their very nature promotes DRY code, and allows abstraction of the complex underlying behaviour to view the overarching design of the AI.
A behaviour tree functions like any regular tree structure. It is a recursively defined structure consisting of a root node with any number of children which are also nodes. A node without children is traditionally called a leaf, however in the implementation of a behaviour tree, they are referred to as Tasks since they are the nodes that contain the functions that will alter the AI's state.
A leaf's parents act as conditional logic and flow control, reacting to the current state of the game world and executing certain children under certain conditions. Trees are executed starting at the root node, and working through the children moving from left to right.

Since this data structure provides functionality for flow, conditional logic and nodes which perform functions, the resulting structure is a very powerful tool almost like a scripting language within java itself. Since every item in a tree is a node, a sub-tree can be a child of a parent node, leading to the ability to build up complex trees and reusing certain sub-tree sections. These re-usable sub-trees are known as behaviours and act like the brain of an AI.
To enable different nodes to communicate with each other, a map object can be passed to each node so that information about the state of the world and the AI can be stored this map is referred to as the context of a tree.

The following sections will detail the design and implementation of this concept, along with some points for improvements going forward.

2.7.2 Design

Every node in the tree returns a Status of it's execution. It can be one of three values: Success, Failure, or Running. The Running status is returned if the node needs to be re-evaluated on the next game tick without being reset. These return values can be used by a parent node to alter the flow of a behaviour tree's execution.
The return Status can be used by a node to signal that it cannot execute, this can be used in error handling. Any time that the context is accessed, first it is checked to see if it has the element stored. If it does not, the node will either return Failure or use a default value. Usually the node will only return Failure if it is a compulsory value that is missing. (see section 2.7.5 for definition of a compulsory value)

Listing 2.25: Top level behaviour tree of a Drone in Tanktrum

```
new Selector (
    Conditionals.healthBelowThreshold(
            selfDestructTree(100)
    ),
    chaseTree()
)
```

Composite Nodes

Composite nodes are a form of of flow control node. The main two types are the Selector and the Sequence.

Selector This node will execute it's children starting from left and moving right, until one of them returns either Running or Success. When this happens, the Selector returns the child's result to it's parent.
If the child returns Failure, the Selector will simply execute the next child in it's sequence and repeats this process. If all of it's children return Failure, the Selector returns Failure.

Sequence This node will execute it's children from left to right until one of them returns Failure or Running. Then it will return this value to it's parent.

Conditional Nodes

Conditional nodes only execute their children if a certain condition is met, for example if the AI's health is below a certain value.

Decorator Nodes

Decorator nodes alter the result returned by it's child. An example would be a Succeeder, which will always return Success to it's parent regardless of its child's result.

2.7.3 Implementation

2.7.4 BNode

Every node in the tree structure extends the BNode class. It contains the reset and process methods called during execution of the tree at runtime, as well as a construct method which is called on object construction to allow additional fields to be initialised.

```
public abstract class BNode {
private Status status;

public BNode(){
    status = Status.READY;
    construct();
}

public void construct(){

}

public void reset(BContext context){
    this.status = Status.READY;
}

public abstract Status process(BContext context);

...
}
```

Composite Nodes

Both Sequence and Selector have a lot of features in common and so they are both extending a class called CompositeBNode. The only difference between the two is that Selector continues execution if a child returns Failure, Sequence continues execution if a child returns Success. This property means a Status argument is passed to CompositeBNode when the node is instantiated to signify which value it should continue execution on.

Listing 2.26: CompositeBnode process method

```
@Override
public Status process(BContext context) {

    if (getStatus() != Status.RUNNING)
        reset(context);

    while(!executionSequence.isEmpty()){
        //fetch child, but dont remove from queue
        BNode child = executionSequence.peek();

        Status childStatus = child.process(context);

        if (childStatus == continueExecutionStatus){
            //if the child returned the continueExecution status,
            //remove it from the queue.
            executionSequence.poll();

        } else {
            //if the child returns an unfavourable result
            //return it's status and halt execution
            setStatus(childStatus);
            return getStatus();
        }
    }
    //all children were executed successfully
    setStatus(continueExecutionStatus);
    return getStatus();
}
```

Conditional Nodes

Conditional nodes only have one child, since they either execute the child or they return Failure. The condition for whether they execute the child or not is passed in as a function that takes the context as an argument and returns a boolean.

Listing 2.27: ConditionalBNode process method

```
@Override
public Status process(BContext context) {
    //if the eval for executing the child is true, execute the child
    if (condition.eval(context)){

        if (child.getStatus() != RUNNING){
            //if child isn't running, reset it
            child.reset(context);
        }

        Status status = child.process(context);
        setStatus(status);
        return status;
    }
    setStatus(FAILURE);
    return FAILURE;
}
```

Decorator Nodes

Decorator nodes alter the result of it's child. There is no specific class that this type of node can draw from, and so some basic types were created.

RepeatN This node will return Running for a given number of ticks, until after which it will return Success. It does not care what the child returns.

2.7.5 BContext

Each node takes a parameter called context when it is executed. This context is an object containing two object maps. One for fields that a child cannot execute without (for example a reference to the controlled entity itself) named the compulsory map. The other is used as a communication pathway for nodes to share results with each other

Listing 2.28: BContext constructor and fields

```
private Map<BKeyType, Object> compulsoryFields;

private Map<String, Object> variables;

public BContext(){
    this.compulsoryFields = new HashMap<>();
    this.variables = new HashMap<>();
}
```

Bundled Tasks

A number of tasks in this framework contain implementations of steering behaviours outlined in the paper "Steering Behaviours For Autonomous Characters"[7] and as such I will not be discussing the implementations here as they closely follow the outlines in the paper.

Bundled Behaviours

To demonstrate the power of behaviour trees, a number of behaviours were implemented in the framework. Outlined in listing 2.29 is the chase behaviour, in which the AI will hunt down the player, and then shoot at the player once it arrives. Due to the high abstraction, it is clear to see what each section does. Notice that the shootBehaviour is itself a behaviour, and this tree doesn't care how it executes.
Note: A ForgetfulSequence resets from the first child every tick, even if one was Running last tick.

Listing 2.29: Chase Behaviour Tree

```
new Sequence(
    Tasks.findMeshAdjacentToPlayer(),
    new RunTillArrived(
        new ForgetfulSequence(
            new Succeeder(
                Conditionals.canSeePlayer(
                    shootBehaviour()
                )
            ),
            travelTo(false)
        )
    ),
    new Selector(
        Conditionals.canSeePlayer(
            new RepeatN(2,
                new Sequence(
                    new Succeeder(
                        shootBehaviour()
                    ),
                    Tasks.doNothing(25)
                )
            )
        )
    )
)
```

2.7.6 Afterthoughts

Moving forward, I would like to work on the efficiency of some of the implemented tasks, notably the course correction Task. Due to the nature of it's execution, every game object must be checked for a collision with the sight rectangle. This implementation is functional for smaller numbers of AI but as the number increases, performance is affected dramatically. Incorporating this sight rectangle into the collision detection step, and setting a flag if it is found to be colliding with an entity other than the AI owner would be a more sensible approach.

As a whole however, the current implementation of the more abstract nodes such as the composites and conditionals, gives the user the freedom to override any part of the tree nodes for their own version if they wish to. The variety of bundled node types allows the user to quickly and easily construct complex behaviours for their AI.

2.8 Game Maps

2.8.1 Introduction

This section will cover the classes that aid the player in loading a game level. The navmesh generator takes a 2D list of integers comprised of the blocked/unblocked cells. As you will see in section 2.8.2, the user provides an image as the layout of the game map. This is interpreted into the list required by the navmesh generator.

2.8.2 MapReader class

Design

This purpose of this class is to interpret an image as a map layout file. This method was chosen as it allows the user to quickly construct a level without having to manually insert each wall and door into the map via code.

Implementation

Android provides an API for retrieving files from the assets directory in the phone that the Operating System is running on, the image is loaded into memory as a 2D list which is then iterated on. It's simply the case of assigning certain pixel colors different meaning. For this implementation, I have chosen walls to be black (#000000), and doors to be blue (#0000ff).
The secondary function that this class provides is to determine the orientation of the door based on the surrounding walls. For example the orientation of a door is different if the walls are on the left and right, or if the walls are on the top and bottom (Directions are relative to the image). Any other configuration of walls is considered invalid and will result in the door being skipped and not read in.

2.8.3 MapConstructor class

Design

This class is the main controller for loading a level in the framework. It contains the method calls to MapReader and MeshConstructor, and handles the initialization of walls and doors into game objects.

The wall cells in the image will each be one cell in size if they are directly turned into game objects, so there needs to be an algorithm to join sections of the wall together to form the largest convex shapes possible to increase efficiency.

Implementation

Doors In order to generate the doors, it requires other information about them other than what can be interpreted from the MapReader so a DoorBuilder class. This class acts as a holder for all the data needed to spawn a door in. The MapConstructor class requires that the name, tile the it lies in, and the locked status of the door is already specified before it creates the door objects.
After executing MapReader, the DoorBuilder class will contain all the needed information to create a door game object, however there remains the task of finding the meshpolygon that the door lies in, as well as the polygons on either side of the door. This is so that the door can alter the navmesh graph when it is locked/unlocked (see listing 2.30).

Listing 2.30: Main loop of constructDoors method in MapConstructor

```
for (DoorBuilder doorBuilder : mapReader.getDoorSpawns()){

    Tile sideSets = null;
    int doorSet = 0;

    for (MeshPolygon meshSet : map.getRootMeshPolygons().values()){

        if (meshSet.getBoundingBox().intersecting(doorBuilder.getCenter())){
            //if the door is in this polygon


            doorSet = meshSet.getId();
            List<Edge<Integer>> neighbours = meshConstructor.getMeshGraph()
                    .getNode(doorSet).getConnections();

            //if the door polygon has more than two neighbours,
            //it is not a valid door configuration
            //as a door has to have exactly two sides open in order to pass through
            //more than two sides open means the door is
            //pointless as it can be circumvented
            if (neighbours.size() == 2) {
                sideSets = new Tile(neighbours.get(0).traverse().getNodeID(),
                        neighbours.get(1).traverse().getNodeID());
            }
        }
    }

    //if the polygons on either side of the door could be determined,
    //create the door object
    if (sideSets != null && doorSet != 0) {
        Door door = new Door(doorBuilder.getName(), doorBuilder.getCenter(),
                    tileSize, tileSize, doorBuilder.isLocked(),
                    doorBuilder.isVertical(), ColorScheme.DOOR_COLOR,
                    doorSet, sideSets);
        doors.add(door);
    }
}
```

Walls In order to reduce the number of wall objects created in the game world, we can perform a walk around each cell of wall type, and merge consecutive cells together provided they share a common direction to preserve the convex shape needed for collision detection.

Listing 2.31: Wall Optimization Algorithm loop

```
List<Tile> currentWall;
while (!wallsToCheck.isEmpty()){
    currentTile = wallsToCheck.poll();

    currentWall = new ArrayList<>();
    currentWall.add(currentTile);

    List<Tile> neighbours = getNeighbours(currentTile, tileList);

    if (neighbours.get(0) != null || neighbours.get(2) != null){
        //the wall has at least one vertical neighbour,
        //walk in both vertical directions adding all wall cells found to the currentWall
        currentWall.addAll(walkInDirection(currentTile, new Tile(0, 1), tileList));
        currentWall.addAll(walkInDirection(currentTile, new Tile(0, -1), tileList));

        //sort by y coordinate so the top and bottom coordinates can be
        //determined
        currentWall.sort(Comparator.comparingInt(Tile::getY));

    } else if (neighbours.get(1) != null || neighbours.get(3) != null){
        //the wall has at least one horizontal neighbour,
        //walk in both horizontal directions adding all wall cells
        //found to the currentWall
        currentWall.addAll(walkInDirection(currentTile, new Tile(1, 0), tileList));
        currentWall.addAll(walkInDirection(currentTile, new Tile(-1, 0), tileList));

        //sort by x coordinate so the top and bottom coordinates can be determined
        currentWall.sort(Comparator.comparingInt(Tile::getX));
    }

    //create the wall using the cells contained in currentWall
    //passing the first and last cells as parameters to
    //define the top left and bottom right of the wall.
    //as the coordinates reference the top left corner of a cell,
    //the last cell needs to be incremented
    //in order to meet the bottom right point of the cell
    walls.add(createWall(new Point(currentWall.get(0)), new Point(currentWall.
            get(currentWall.size()-1).add(1,1))));

    //remove any checked wall tiles from the wallsToCheck queue
    for (Tile tile : currentWall){
        wallsToCheck.remove(tile);
    }
}
```

2.8.4 Afterthoughts

This section performs as designed, allowing the user to quickly construct large maps without writing any Java code for the walls, and minimal code for the doors. However, an improvement could be made to the method in which the image is read in. For really large images or low end phones, the memory could easily not have enough space to store the entire image at one time. Buffers should be used to only inspect the part of the image file that is needed at the current step of the algorithm.

2.9 Input

2.9.1 Introduction

Android is a touch screen based platform, and supports multiple active inputs at one time. In order for developers to identify which input is which from one event to another, the Android API assigns an id to each active "pointer".
The Android Canvas class which I am using for rendering does not have inbuilt clickable UI elements like buttons, so I had to implement them myself. The interface that these custom UI elements use, can be found in section 2.3.5.

In order to receive input from the user to move the player around, I implemented a Thumbstick class which is described in section 2.9.2.

2.9.2 Thumbstick

In order for the player to move around the game world, they need a method of providing input. A classic method is to use a thumbstick, as it allows fully continuous input. That is, the player can move in any direction they wish, just by moving the thumbstick to point in that direction. In contrast, using a keyboard arrows only allow 4 directions of movement.

2.9.3 Implementation

The thumbstick has two states: neutral and active. In the neutral state (see figure 2.17a) the thumbstick is not receiving any input. In the active state (see figure 2.17b), the thumbstick is receiving input.

The desired movement vector is calculated by taking the position of the thumbstick's neutral center and the center of the active position, truncating the length if it exceeds the specified max length.
This vector can then be directly passed to the player object during a game tick. However since the thread with the event listeners on is not in sync with the game loop thread, When it is time for the game loop to register input from the player, the loop will fetch the current vector being input by the player.

Listing 2.32: Thumbstick setActivePosition Method

```
public void setActivePosition(Point newActivePosition) {

    if (receivingInput){
        Vector newInputVector = new Vector(neutralPosition, newActivePosition);

        float proportionOfLengths = newInputVector.getLength() / maxRadius;

        if (Float.compare(proportionOfLengths, 1f) > 0){
            //shrink input vector to length of max radius if too long
            newInputVector = newInputVector.sMult(1f / proportionOfLengths);
        }
        this.inputVector = newInputVector;
    }
}
```

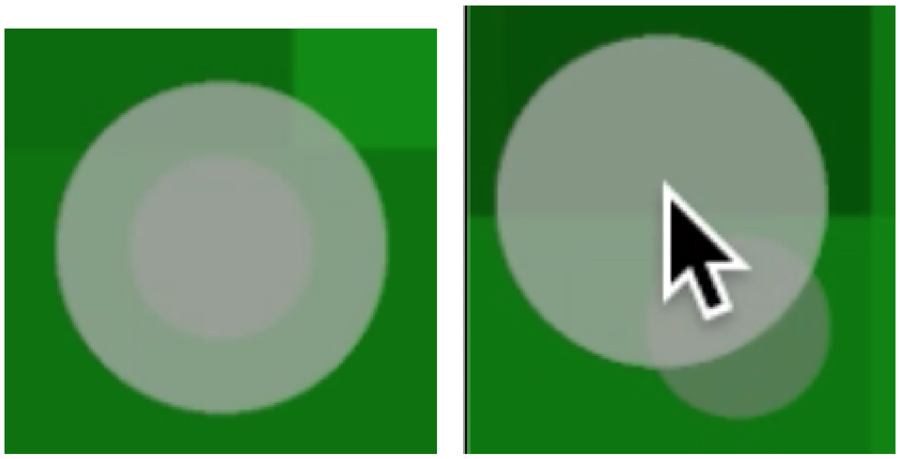

(a) Thumbstick not receiving any input (b) Thumbstick in the active position

Figure 2.17: Thumbstick in a neutral state and receiving input

2.10 Popups

2.10.1 Introduction

This framework does not directly include a rendering engine. Renderable game objects have draw functions implemented so that they can easily be slotted into a game's rendering engine however.

Despite this, an important aspect of game design is to communicate with the player whilst they are playing. The player needs to be made aware if the level ends, or if the level is paused, these are taken care of in a custom built popup system, described in section 2.10.2.

2.10.2 Implementation

The idea of this system is that the user designs a screen to be shown, and the system will take care of placement within the screen dimensions provided. The user specifies a vertical position that he would like to have the elements, and the Popup class orders all the elements according to their vertical position field.

PopUpElement Interface

This interface contains all the methods needed to have a custom object be compatible with the Popup system.

As seen in listing 2.33, a static method for bounding the vertical placement is provided. This method is called by the Popup class when it is sorting the elements it controls.

Listing 2.33: PopUpElement Interface

```
public interface PopUpElement {

    int getHeight();
    int getWidth();
    void setCenter(Point point);
    int getVerticalPosition();
    void draw(Canvas canvas, Point point, Paint textPaint);

    static int boundVerticalPosition(int verticalPosition){
        if (verticalPosition < 0)
            verticalPosition = 0;
        else if (verticalPosition > 100)
            verticalPosition = 100;

        return verticalPosition;
    }
}
```

Popup Class

A popup is placed on a portion of the screen, not necessarily the entire screen. It is given a width and a height, and a list of elements to place in it's window.
The elements are placed at the vertical position stored in it's field. This position is treated as a percent of how far down the popup should the element appear. For example, a vertical position of 20 will place the element 20% of the way down the pop up.

Listing 2.34: Popup Class constructor and fields

```
private List<PopUpElement> elements;
private boolean show;
private boolean pauseOnShow;
private Paint textPaint;

private IPause pauseFunction;

private Rectangle background;

public Popup(Point center, int width, int height, boolean pauseOnShow,
            List<PopUpElement> elements, IPause pauseFunction){
    this.elements = new ArrayList<>();

    this.background = new Rectangle(center, width, height, Color.LTGRAY);

    if (elements == null || elements.size() == 0)
        throw new IllegalArgumentException("Popup constructor requires at least"+
                    "one element");

    elements.sort(Comparator.comparingInt(PopUpElement::getVerticalPosition));

    this.elements = elements;

    this.elements.forEach(element ->
            element.setCenter(new Point(center.getX(), center.getY() - (height/2f)
                    + height * (element.getVerticalPosition()/100f))));

    show = false;
    this.pauseOnShow = pauseOnShow;
    textPaint = RenderingTools.initPaintForText(Color.DKGRAY, 30, Paint.Align.CENTER);

    this.pauseFunction = pauseFunction;
}
```

2.11 Shapes

2.11.1 Introduction

In this framework, both the rendering and the collision detection use the same shape model for their operations. For a description of the Shape interface that all shape classes must comply with, see section 2.3.4.

The framework comes bundled with some basic shapes, and an extensive Polygon class that is easily extended to form new n-sided polygons. The nature of the collision detection algorithm in use means that it is difficult to capture both polygons and circles under the same umbrella. This is why the two main Shape classes are Polygon and Circle.

2.11.2 Circle

ICircle

It is important that an object that has a circular shape implements the ICircle interface (For example, projectile). This is because the collision manager makes use of 'instance of' statements to determine if a shape is circular or not. This interface provides the methods needed to analyze a circle for collisions (See listing 2.35).

Listing 2.35: ICircle Interface

```
public interface ICircle {
    Point getCenter();
    float getRadius();

    Circle getCircle();
}
```

Circle

This class implements shape and therefore has all of the methods outlined in section 2.3.4, in addition to the methods in ICircle. The constructor can be seen in listing 2.36.

Listing 2.36: Circle Constructor

```
private float radius;
private Point center;

...

public Circle(Point center, float radius, int color){
    this.radius = radius;
    this.center = center;

    ...
}
```

It is worth noting that the getVertices() and getVertices(Point offset) methods seen in listing 2.37 use the bounding box and not an approximation of the circle. These methods are only called on circular objects that do not implement ICircle so that the object could still be used with the collision methods, but not very accurately.

Listing 2.37: Circle getVertices Methods

```
@Override
public Point[] getVertices() {
    return getBoundingBox().getCollisionVertices();
}

@Override
public Point[] getVertices(Point offset) {
    Point[] vertices = getBoundingBox().getCollisionVertices();

    for (int i = 0; i < vertices.length; i++){
        vertices[i] = vertices[i].add(offset);
    }

    return vertices;
}
```

2.11.3 Polygon

Introduction

The Polygon class (not to be confused with the MeshPolygon class) is an abstract class that is designed as a starting platform for any n-sided polygons. It contains implementations for most of the Shape Interface methods. The only methods it does not handle are the methods relating to rendering, and the setter for the forward unit vector. This means that it is very simple to create a new type of Polygon.

The Forward Unit Vector

The forward unit vector is not implemented by Polygon as it is how Polygon knows which way the object is facing. As you will see in the Rectangle class, this is easily defined as initially when the shape is first created, it is oriented with the forward vector facing up which in the left handed coordinate system we are working in, can be represented by the unit vector [0,-1].

It is advisable to store the angle between this unit vector and the vertex vector closest to it, so that the forward vector can be set by rotating the selected vertex vector by the stored 'primary' angle. See listing 2.38 for the implemented method in Rectangle.

Listing 2.38: Rectangle's setForwardUnitVector() method

```
@Override
protected void setForwardUnitVector() {
    this.forwardUnitVector = vertexVectors[0].rotate((float) Math.cos(primaryAngle),
            (float) Math.sin(primaryAngle));
}
```

Constructing a Polygon

There is only one constructor available for the Polygon class (See listing 2.39). Most of the arguments are easy to recognize except for vertexVectors.

The vertex vectors field is what allows the Polygon implementations to function regardless of the number of edges it may have. It consists of a vector for each vertex, with the tail located at the center of the object, and the head located at the vertex itself. This way of storing the vertices is how the rotations and translations are completed.
The reason that the 'center' field is not protected like with the other fields a sub-class might like to access, is so the center cannot be altered without also altering the vertex vectors. This is done via the translate methods.

Listing 2.39: Polygon constructor and fields

```
private ShapeIdentifier shapeIdentifier;
protected Vector[] vertexVectors;
protected Vector forwardUnitVector;
private Point center;

protected Polygon(Point center, Vector[] vertexVectors, ShapeIdentifier shapeIdentifier){
    this.vertexVectors = vertexVectors;
    this.center = center;

    this.forwardUnitVector = new Vector(0,-1);

    this.shapeIdentifier = shapeIdentifier;
}
```

Implemented Shape Methods

In listing 2.40, I have included the methods which involve the vertex vectors. These methods have been tested on polygons with upto 8 sides, and have worked without fault.

Listing 2.40: Polygon Implemented Shape Methods

```
@Override
public void rotate(float angle){
    float cos = (float) Math.cos(angle);
    float sin = (float) Math.sin(angle);

    //iterate through the vertex vectors, and rotate each one
    for (int i = 0; i < vertexVectors.length; i++){
        vertexVectors[i] = vertexVectors[i].rotate(cos, sin);
    }

    setForwardUnitVector();
}

@Override
public void translate(Vector movementVector){
    Point amountToTranslate = movementVector.getRelativeToTailPoint();

    center = center.add(amountToTranslate);
    //iterate through the vertex vectors, and rotate each one
    for (int i = 0; i < vertexVectors.length; i++){
        vertexVectors[i] = vertexVectors[i].translate(amountToTranslate);
    }
}

public Point[] getVertices() {
    Point[] vertices = new Point[vertexVectors.length];
    //forms an array containing the heads of the vertex vectors,
    //which form the vertices of the shape.
    for (int i = 0; i < vertexVectors.length; i++){
        vertices[i] = vertexVectors[i].getHead();
    }
    return vertices;
}
```

Rectangle

As an example of how the Polygon class is implemented, listing will show the Rectangle class. The only methods that had to be implemented were the rendering methods, and setForwardUnitVector() which is dependent on the Polygon.

Listing 2.41: Rectangle class

```
private float primaryAngle;

private Paint paint;
private int defaultColor;
private int currentColor;

public Rectangle(Point center, Vector[] vertexVectors, int colorValue){
    super(center, vertexVectors, ShapeIdentifier.RECTANGLE);

    this.primaryAngle = Vector.calculateAngleBetweenVectors(vertexVectors[0],
            this.forwardUnitVector);

    this.paint = new Paint();

    this.defaultColor = colorValue;
    this.currentColor = colorValue;
}

public Rectangle(Point center, float width, float height, int colorValue){
    this(center, new Vector[]{new Vector(center,
            center.add(new Point(-width/2f, -height/2f))),
            new Vector(center, center.add(new Point(width/2f, -height/2f))),
            new Vector(center, center.add(new Point(width/2f, height/2f))),
            new Vector(center, center.add(new Point(-width/2f, height/2f)))}, colorValue);
}

public Rectangle(Point center, Point topLeft, Point bottomRight, int colorValue){
    this(center, new Vector[]{new Vector(center, topLeft),
            new Vector(center, new Point(bottomRight.getX(), topLeft.getY())),
            new Vector(center, bottomRight),
            new Vector(center, new Point(topLeft.getX(), bottomRight.getY()))},
            colorValue);
}

@Override
public int getColor() {
    return defaultColor;
}

@Override
protected void setForwardUnitVector() {
    this.forwardUnitVector = vertexVectors[0].rotate((float) Math.cos(primaryAngle),
            (float) Math.sin(primaryAngle));
}

@Override
public void draw(Canvas canvas, Point renderOffset, float secondsSinceLastRender,
            boolean renderEntityName) {

    paint.setColor(currentColor);
    canvas.drawPath(getPathWithPoints(getVertices(renderOffset)), paint);
}

@Override
public void revertToDefaultColor() {
    this.currentColor = defaultColor;
}

@Override
public void setColor(int color) {
    this.currentColor = color;
}
```

2.12 Entities

2.12.1 Entity

Everything in the game world should be uniquely identifiable and this is why (at least in the objects specified in the framework), everything that appears in the game world extends the Entity class. This isn't a large class, it only has one field and that is a String for it's name. In the utils package there is a UniqueId class, and an implementing game should use this class to ensure that entities in the world have unique ids. These ids are also used should the Entity need to be hashed.

2.12.2 Moveable Entity

An entity that is able to move, should extend this class. This class contains all of the methods needed to apply forces, rotate and translate an entity with adjustable maximum values for force limits.
Any system in the game can exert a force on a moveable entity, for example the collision detection system, or the movement input system. Each tick, the moveable entity will calculate the acceleration caused by the resultant of all forces exerted on it and apply this acceleration to itself. See listing 2.42 for the implementation of the applyMovementForces method.

Listing 2.42: MoveableEntity applyMovementForces

```
public void applyMovementForces(float secondsSinceLastGameTick){
    //if the speed drops below a value of 3, then set it to zero.
    //A speed of 3 in this context means 3 pixels per second which is
    //basically not moving.
    if (velocity.getLength() < 3f)
        velocity = Vector.ZERO_VECTOR;

    //the requested movement vector is only used for the player, but
    //this segment should also execute if the entity is an AI
    if (!getRequestedMovementVector().equals(Vector.ZERO_VECTOR) ||
                this instanceof IAIEntity) {
        Vector movementForce = Vector.ZERO_VECTOR;

        //if this is not an ai entity, then apply the requested movement
        //vector force scaled up an amount
        if (!(this instanceof IAIEntity)) {
            movementForce = getRequestedMovementVector().sMult(100);

        }
        //add any external forces applied on this entity to the movement force
        for (Vector force : extraForces){
            movementForce = movementForce.vAdd(force);
        }

        if (movementForce.equals(Vector.ZERO_VECTOR)) {
            //if no force is being applied, decay the existing velocity
            velocity = velocity.sMult(0.25f);
        } else {

            //newtons second law F = ma or in this case rearranged to a= F/m
            Vector acc = movementForce.sMult(1f/mass);

            //truncate the acceleration
            if (acc.getLength() > maxAcceleration) {
                acc = acc.setLength(maxAcceleration);
            }

            //add acceleration to velocity
            velocity = velocity.vAdd(acc);

            float max = getMaxSpeed();

            //truncate velocity
            if (velocity.getLength() > max)
                velocity = velocity.setLength(max);
        }

    } else {
        //no inputted movement vector so decay speed
        velocity = velocity.sMult(0.25f);
    }

    //scale the velocity based on the time passed since the last tick
    Vector v = velocity.sMult(secondsSinceLastGameTick);

    //translate the shape
    shape.translate(v);

    //rotate body in line with movement

    float angularVelocity = getAngularVelocity(v.getUnitVector(),
        secondsSinceLastGameTick, getShape().getForwardUnitVector());

    //truncate angular velocity
    if (angularVelocity > maxAngularVelocity)
        angularVelocity = maxAngularVelocity;

    //rotate the shape
    getShape().rotate(angularVelocity);
}
```

2.13 Evaluation

2.13.1 Fulfillment of the Specification

This framework fulfills the majority of the requirements outlined in the specification in section 2.1. However, it does not meet the following requirements:

- The navigation mesh does not partition on each game tick, as I could not find an efficient way to handle the partitioning using the rectangular axis-aligned style of the navmesh. Had I implemented the navmesh using triangulation, this would be a trivial task. However, to ensure that AI entities do not collide with objects not in the grid, it can call a function in the CollisionTools class that will provide a steering axis using an implementation of the obstacle avoidance behaviour [7].
- There is no separate or inbuilt program for the user to draw maps with at this stage, they will have to use an external program such as https://www.piskelapp.com. This was a low priority requirement and so implementation was pushed back as other sections overran on time.

2.13.2 Next Steps

I am happy with the majority of the framework, however moving forward, the focus will be to implement the remaining requirements first of all. This will mean that the framework can now handle any required data structures internally meaning the ease of use for the user will increase.
Another area for improvement is the flaw in the separating axis theorem outlined in section 2.4.3.

Chapter 3

Developing Tanktrum Using The Framework

3.1 Introduction

This chapter will detail the development of a 2D top-down twin-stick shooter called Tanktrum. The player will take control of a tank and defeat waves of enemies.
The objective of developing this game is to show the functionality of the framework created in the previous chapter. The framework has most of the features that will be present in the final game already, such as the navigation mesh generator, collision detection tools, and behaviour trees for the AI. However, there is still a substantial amount of work needed in order for the game to be complete.

Among the remaining features are a rendering engine, the game logic and loop, the collision manager, and a method for storing data about levels.

3.2 Specification

The game I plan to develop is a 2D top-down twin-stick shooter game. The player will take control of a tank and fight off waves of enemies.

3.2.1 Requirements

Input & UI

1. The player will provide input via two joysticks. One for moving the tank around the world, and the other for rotating the tank turret.
2. There is no dedicated shoot button, instead the tank shoots automatically when the player is aiming the turret.
3. The user can restart or leave the level at any time via the pause menu.
4. Upon launch, the user will be presented with a level select screen.
5. If an entity in game takes damage, a health bar will appear for 3 seconds displaying the amount of health remaining.

AI

There will be three types of AI, a drone, a turret, and a healer. Each of them will have a unique behaviour tree, and will react to the player differently. They should have a change of dropping a health pickup when they die, and a small chance of causing an explosion potentially damaging the player.

Drone The drone will be the most common enemy type. It should be relatively easy to defeat, have a small amount of health, and low damage bullets. The drone should circle the player, shooting at the player as it does so.

Turret The turret will be a moving platform heavy tank which moves incredibly slowly. They will have a large amount of health, and high damage bullets. The turret should hunt down the player and then stop moving in order to fire a shot.

Healer The healer will not engage the player in combat, but instead will hide from the player until one of its ally's health becomes low. At which point it will heal them up. The player should have an incentive to attack these targets first.

AI Controllers

The AI will control their own movement and decision making with their behaviour trees, but there will be a controller active in the level which will choose what AI to spawn and where.

Health Pickups

Health pickups may be scattered throughout the level so that players have a way of evading death for a short while. This should be balanced so the player isn't carefree about getting damaged.

Map File Format

In order to store the spawn locations for entities, and other parameters that are relevant to a level, I will use the json file format to provide an easy way to access fields in the file by name.

3.3 Game Loop

3.3.1 Introduction

The game loop is the heart of any game. It governs how often the game will tick, and how often a render tick will occur. The loop implemented in this game is one which will tick the game logic at a constant 25 ticks per second, and will render as often as it can with the remaining time before the next game tick is scheduled. [11],[12]

3.3.2 Implementation

Listing 3.1 shows how the game loop has been implemented in Tanktrum.

Listing 3.1: Game Loop Implementation

```
//the maximum number of game ticks that will be executed in a row
//without performing a render tick
int maxSkipTick = 5;
//the total tick time for a game and render tick
//set to be 40ms, which means the game ticks 25 times a second
int gameTickTimeStepInMillis = 40;
long nextScheduledGameTick = System.currentTimeMillis();

while (running){

    int consecutiveGameTicks = 0;

    //only tick the game if it isnt paused
    if (!levelState.isPaused()) {

        currentGameTick = System.currentTimeMillis();

        while (currentGameTick > nextScheduledGameTick && consecutiveGameTicks <
                maxSkipTick) {

            //read input from each thumbstick.
            levelState.getPlayer().setRequestedMovementVector(inputManager.getThumbstick()
                    .getMovementVector());
            levelState.getPlayer().setRotationVector(inputManager.getRotationThumbstick()
                    .getMovementVector());

            //remove any edges into polygons that have a certain percentage of their
            //area blocked
            //and map moveable entities into the mesh
            computeMeshBlockages();
            //tick the player and any explosions/bullets/other entities
            entityController.update(gameTickTimeStepInMillis / 1000f);

            //tick the ai
            levelState.getAiManager().update(gameTickTimeStepInMillis / 1000f);

            //check collisions
            collisionManager.checkCollisions();

            //schedule the next game tick
            nextScheduledGameTick += gameTickTimeStepInMillis;
            consecutiveGameTicks++;

            //used for first time that the level is ticked to force a game update before
            //the renderer ticks
            levelState.setReadyToRender(true);

            //check level end conditions
            checkIfLevelEnded();
        }
    } else {
        nextScheduledGameTick = System.currentTimeMillis();
    }

    if (!levelState.isPaused() && levelState.isReadyToRender()) {
        //calculate time passed since the last game tick
        //used for render interpolation
        float timeSinceLastGameTick = (System.currentTimeMillis() +
                gameTickTimeStepInMillis - nextScheduledGameTick) / 1000f;
        levelView.draw(renderFPSMonitor.getFPSToDisplayAndUpdate(),
                gameFPSMonitor.getFpsToDisplay(), timeSinceLastGameTick);

    } else if (levelState.isPaused() && levelState.isReadyToRender()){
        //special condition for if the level is paused,
        //passes 0 for time passed since last tick to stop interpolating
        levelView.draw(renderFPSMonitor.getFPSToDisplayAndUpdate(),
        gameFPSMonitor.getFpsToDisplay(), 0f);
    }
}
```

3.3.3 Afterthoughts

The implemented game loop handles overrunning game ticks fluidly by skipping rendering ticks, however if the render ticks per second drops below around 5, the game ticks per second will start to lower as well [12]. There are a few factors that could cause this, rendering taking a long time, intense calculations causing the game ticks to overrun often, or hardware that isn't capable of keeping the tick rate above a certain margin. To improve this game loop, alterations to the quality of drawn items on screen could be made to lower the complexity of a render tick, potentially raising the overall tick rate. However, since the rendering engine is using Android's built in Canvas functionality, there are limits on how much the quality can change (see section 3.6 for more on the rendering engine).

3.4 Collision Manager

3.4.1 Introduction

The algorithms for detecting collisions are already implemented in the framework, so the collision manager only has to apply whichever ones it needs depending on the situation.

3.4.2 Design

Each game tick, the game loop will call the checkCollisions() function on the collision manager. Listing 3.2 will show how this executes. As most of the implementation is already in the framework, the only additional efficiency tweak that has been added is to not evaluate a pair of objects that are both immovable (walls), or are both not solid (e.g. interaction fields for doors). This is because the types of objects that match that condition have been placed there by the level designer, and are therefore already in the correct positions.

Listing 3.2: Collision Manager Pseudo-Code

```
FOR each spatial bin in the spatialBins list
    IF bin has less than two elements in
        CONTINUE to next bin
    ENDIF

    FOR each object in the bin (loop 1)
        FOR each object in the bin starting from the second object (loop 2)

            IF both objects cannot move OR both objects are not solid
                CONTINUE to next object in loop 2
            ENDIF

            IF one of the objects is not solid
                //the following if blocks address the non solid object
                IF one is an explosion
                    //explosions are known to be circular in the game
                    call collision check between circle and the solid object

                    IF colliding
                        damage solid object
                    ENDIF
                ELSE IF one is a projectile
                    //projectiles are known to be circular in the game
                    call collision check between circle and the solid object

                    IF colliding
                        damage solid object
                    ENDIF
                ELSE IF one is a door field
                    call collision check between two collidables

                    IF colliding AND door is unlocked
                        mark door for opening
                    ENDIF
                ELSE IF one is a pickup
                    IF solid object isn't the player
                        CONTINUE to next object in loop2
                    ENDIF

                    call collision check between two collidables

                    IF colliding
                        activate pickup
                    ENDIF
                ENDIF
            ELSE
                //both objects are solid
                collision check two collidables

            ENDIF
        ENDFOR
    ENDFOR
ENDFOR

remove any explosions from the level state
toggle any doors that have been marked
```

3.4.3 Implementation

As the implementation of the main collision body heavily follows the outline set out in listing 3.2, only the resolution of two solids will be covered here. Any "check collision" call is performed with a framework method.

Listing 3.3: Resolution of two solid objects colliding

```java
Point newCenter;

if (firstCollidable.canMove() && secondCollidable.canMove()) {
    //both entities can move

    //amount added to the entity's center to allow reverting to previous state
    Point firstAmountMoved;
    Point secondAmountMoved;

    int firstMass = ((MoveableEntity) firstCollidable).getMass();
    int secondMass = ((MoveableEntity) secondCollidable).getMass();

    if (firstMass == secondMass){
        //if they have the same mass, move both equally
        firstAmountMoved = pushVector.sMult(0.5f).getRelativeToTailPoint();
        secondAmountMoved = pushVector.sMult(-0.5f).getRelativeToTailPoint();
    } else if (firstMass > secondMass){
        //if first has greater mass, only move second
        firstAmountMoved = new Point();
        //since the push vector always aims to push the first entity,
        //it needs to be flipped
        secondAmountMoved = pushVector.sMult(-1).getRelativeToTailPoint();
    } else {
        //second has greater mass, only move first
        firstAmountMoved = pushVector.getRelativeToTailPoint();
        secondAmountMoved = new Point();
    }


    //update object positions with calculated movement amounts
    newCenter = firstCollidable.getCenter().add(firstAmountMoved);
    firstCollidable.setCenter(newCenter);

    newCenter = secondCollidable.getCenter().add(secondAmountMoved);
    secondCollidable.setCenter(newCenter);

} else {
    //only one entity can move

    MoveableEntity moveableCollidable;

    //find out which entity is able to move
    if (firstCollidable.canMove()) {
        moveableCollidable = (MoveableEntity) firstCollidable;

    } else {
```

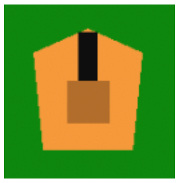


Figure 3.1: Drone Enemy

```
        moveableCollidable = (MoveableEntity) secondCollidable;

        //since the push vector always aims to push the first entity,
        //it needs to be flipped
        pushVector = pushVector.sMult(-1f);
    }

    //move the entity
    Point amountToMove = pushVector.getRelativeToTailPoint();
    moveEntityCenter(moveableCollidable, amountToMove);

    //the bigger the angle at which the moveable entity collides with the immovable,
    //the less force that will be exerted (glancing blow)
    float angle = Vector.calculateAngleBetweenVectors(moveableCollidable.getVelocity()
    .sMult(-1), pushVector);
    float prop = 1 - (angle / (float)Math.PI);

    //add a force to the entity to force them away from the wall
    moveableCollidable.addForce(pushVector.setLength(100).sMult(prop));
}
```

3.5 AI

3.5.1 Introduction

As we are using the framework, we are able to harness the systems implemented within it. The majority of the required tasks such as the steering behaviours and navigation tasks are already implemented. All that is required is to construct the behaviour trees of the enemy types.

3.5.2 AI Enemy Types

Drone

The basic enemy is called a Drone. It's behaviour can be seen in listing 3.4. These types of enemies have low health, and don't deal that much damage to the player. However, they can move quickly and area able to easily maneuver around the player.

Listing 3.4: Drone Behaviour Tree

```
public static BNode droneTree(){
    return new Selector(
            brokenDownTree(),
            chaseTree()
    );
}
```

Turret

The turret is a heavy tank, so it has a lot of health and moves slowly. However, it's bullets deal more damage than the Drones. Upon their death they cause an explosion, damaging any entities in it's radius.

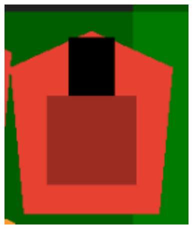


Figure 3.2: Turret Enemy

Listing 3.5: Turret Behaviour Tree

```
public static BNode turretTree(){
    return new Selector(
            brokenDownTree(),
            new Selector(
                    Conditionals.canSeePlayer(
                            delayedShoot(50, 10)
                    ),
                    new Sequence(
                            Tasks.findMeshAdjacentToPlayer(),
                            travelTo(false)
                    )

            )
    );
}
```

3.5.3 AI Sight

Whilst the majority of features are already present in the framework, there is no algorithm for evaluating an AI's line of sight with the player. I have implemented a simple algorithm which casts a ray from the center of the ai to the player, and collision checks the objects in the game world. If an intersection is detected then the line of sight can be considered to be broken.

This approach works well for the number of AI that are in the game world at one time during a level, however it is easy to see how for a large number of AI, in a large game world, this becomes an expensive operation. To combat this, an improvement will be made where the ray between the centers is placed into the spatial-partitioning bins so only relevant objects are evaluated with collision checks. Using a flag I can then notify the AI if their line of sight is obscured. Perhaps even plotting a path to the nearest place where line of sight is not obscured.

3.6 Rendering

3.6.1 Introduction

The framework does not have a rendering engine package in with it, instead it uses the inbuilt Android Canvas utility for drawing objects to the screen. The game objects come pre-loaded with draw functions aimed at this utility, so the rendering engine in this game loops through all the game objects and draws them onto the canvas.

3.6.2 Object Culling

In most games, the part of a tick that consumes the most time is usually the rendering. Games that have large textures on their objects endeavor to reduce the unnecessary drawing of objects that aren't on the screen. One method to do this is to check if an object intersects the visible bounds of the game window, and if it does, then it should be rendered.

I initially implemented one such algorithm for Tanktrum to reduce rendering times as the game worlds grew larger, however can be seen in section 3.6.2, the algorithm actually increased the render times so I removed the algorithm calls. I suspect that this is due to the game objects only having a solid fill texture, instead of having an image mapped onto the surface.

Table 3.1: Table showing the increase in render times as the number of objects on screen increases

Number of Entities on Screen	Average Render Tick (ms)
19	10
30	15
56	15
100	16
160	18

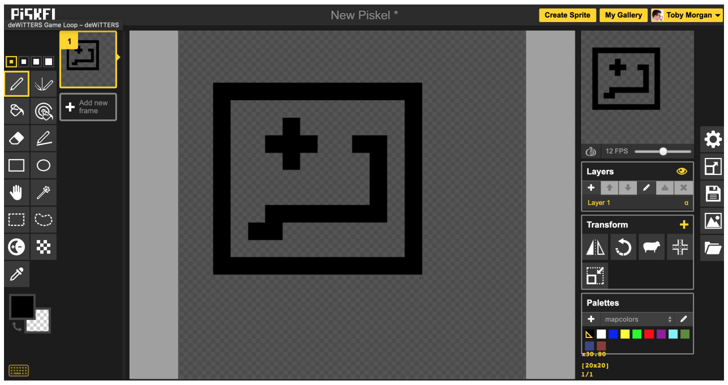


Figure 3.3: The Piskel Interface

Benchmarks

The following benchmarks were conducted on the same control map under the same conditions as in section 2.4.6. The results were obtained by navigating through the map, stopping on sections with different numbers of entities on the screen.

When the benchmarking was conducted without object-culling, despite moving through the map, the same number of elements were drawn to the canvas. There were 144 elements drawn at any given tick, and the average render time was always 11ms. The times for when object culling was turned on can be found in table 3.1.
This result surprised me as I had the object culling algorithm in my game since a very early stage, and to find out that it was having a negative impact on performance was interesting.

3.7 Level Files

3.7.1 Introduction

In order to make use of the framework's tools for mesh generation and spawning, we will need to provide systems for storing and loading levels and maps. The framework provides functionality for reading maps in for mesh generation from images, and so to draw out the maps and placement of the doors, I used https://www.piskelapp.com/ which is a pixel art production platform (See figure 3.3). It allows me to paint pixel by pixel on custom sized images, and was very helpful in producing levels quickly.

Along with the image of the map, I have to specify where to spawn entities, what properties they have, and other information relevant to the level. For this I have used json files as it allows me to serialize information about the game entities easily (see figure 3.6).

3.7.2 MapLoader

This class acts as a parser for reading in json files when loading a level. The package used for parsing from json files and writing to json files can be found at https://mvnrepository.com/artifact/org.json/json. There are other packages that do the same function with incredible speed and efficiency, most notably is jackson json however the performance just isn't needed here. [13] I'm not going to be reading huge json files, so I won't need the buffer capabilities of jackson. Another factor in me choosing org.json is that it is well documented online, so any issues were easy to resolve.

Listing 3.6: Example json file for a game level

```
{
    "player": {
        "osr": {"x": 0, "y": 1},
        "sp": {"x": 10.5, "y": 4.5}
    },

    "doors": [
        {
            "name": "DOOR1",
            "tile": {"x": 10, "y": 5},
            "locked": false
          }
    ],

    "overlords": [
        {
        "lock_on_entry": [
            "DOOR1"
        ],

        "triggers": [
            {"x": 10.5, "y": 6.5}
        ],

        "spawn_points": [
            {"x": 5.5, "y": 8.5},
            {"x": 15.5, "y": 8.5},
            {"x": 7.5, "y": 13.5},
            {"x": 13.5, "y": 13.5}
        ],
        "max_in_level": 4,
        "spawns_in_wave": 6
        }
    ]
}
```

3.7.3 Map

This class holds all of the data about the game map. It implements the interface IMap from the framework in order for it to be passed into the MapConstructor and MapReader classes (see section 2.8).

It serves as a pool for any information about the level entity locations (see listing 3.7) It is built up as the level loads, and is stored in the LevelState class when the level is fully loaded. All of the methods in this class are helper methods for accessing the data stored in the fields, so I won't cover them here.

Listing 3.7: Fields in the Map class

```
private String name;
private String stage;

private int tileSize, widthInTiles, heightInTiles;

private Player player;

private List<AIEntity> enemies;
private List<Wall> walls;

private java.util.Map<String, Door> doors;

private List<Pickup> pickups;

private Graph<Integer> meshGraph;
private HashMap<Integer, MeshPolygon> meshPolygons;

private List<Overlord> overlords;
```

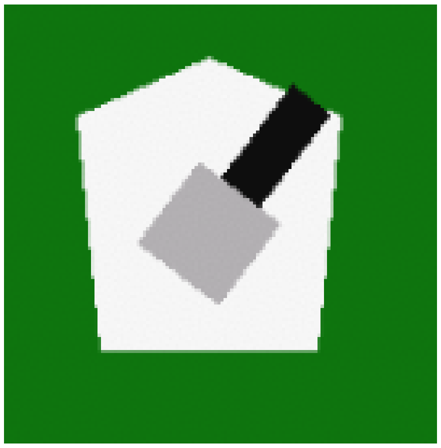
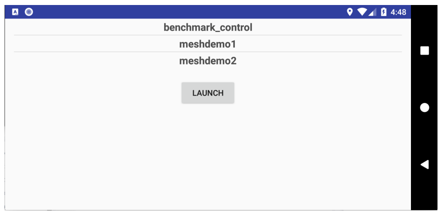


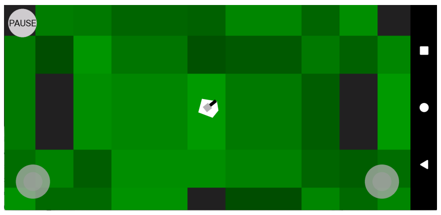
Figure 3.4: The Player

3.8 Game Breakdown

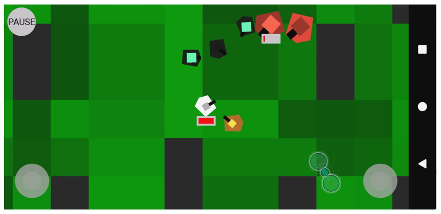
The player (3.4) is placed in a battlefield and is expected to defeat waves of enemy tanks.
There are two different enemy types, Drones (3.1) and Turrets (3.2). They both behave differently so the player will have to keep moving to stay alive.
Enemies have a chance of dropping health pickups when they die, which the player can collect to repair their tank.
The level is won when all enemies are defeated.



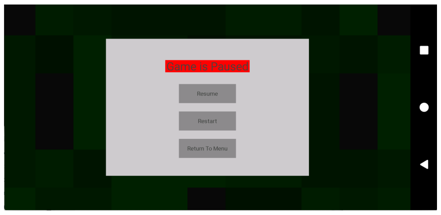
(a) The Level Select Screen



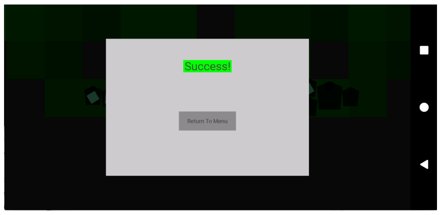
(b) The in game interface



(c) The player engaged in a fight. Greyed out tanks are defeated enemies, green squares are health pickups, the grey and red bars are entity's health bars



(d) The Pause Menu



(e) The Level Completion Screen

Figure 3.5: Final Game Screen-shots

3.9 Evaluation

3.9.1 Fulfillment of Specification

The majority of the specification has been successfully implemented, and the game has all of the elements required. A small change however is that only the turret entity will explode on death. Another addition I made was that if the health of an AI entity drops below a certain threshold, they will be unable to move as they have "broken down".

The requirement that I did not implement was the healer AI type. Due to the time constraints, I decided not to as it does not interact with the player.

3.9.2 Next Steps

Moving forward, I plan to implement the healer AI type, and increase the efficiency of the AI sight algorithm, this will positively influence the game and improve the performance.
Instead of creating more levels, I would like to implement a level generation algorithm. This will constantly present the player with more obstacles to overcome, increasing their enjoyment of the game. Playing the same level over and over may get boring to most players.

A larger feature I would like to implement is that the Overlord AI Controllers could be able to react to the play style of the player, and change the behaviour of spawned entities to raise the challenge to the player.

Chapter 4

Closing Thoughts

4.1 Evaluation

Please find the evaluation of the framework in section 2.13, and the evaluation of Tanktrum in section 3.9.

4.2 Project Management

The amount of time that this project took massively exceeded the time I estimated at the start. As a game framework covers a variety of topics, there were unexpected difficulties that arose from combining them. However, frequent meetings with my supervisor meant that Shan could keep me focused on the aspects that were important.

4.3 Conclusion

I have ended up with an Android game framework and an accompanying game which meets most of the initial specification criteria. Given more time, I would implement the map creation software for the framework so that users can complete the level creation without having to use an external editor like I did whilst developing Tanktrum. However, overall I am satisfied with the implementations I have made, and the project as a whole.

References

[1] 2019. [Online]. Available: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[2] J. Huynh, *Separating Axis Theorem for Oriented Bounding Boxes*, 2008. [Online]. Available: http://www.jkh.me/files/tutorials/Separating%20Axis%20Theorem%20for%20Oriented%20Bounding%20Boxes.pdf

[3] MiiMii1205, "Navigation meshes and pathfinding." [Online]. Available: https://www.gamedev.net/articles/programming/artificial-intelligence/navigation-meshes-and-pathfinding-r4880/

[4] [Online]. Available: https://api.unrealengine.com/udk/Three/NavigationMeshReference.html

[5] S. HERTEL and K. MEHLHORN, *Fast Triangulation of the Plane with Respect to Simple Polygons*, 1985. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0019995885800449#aep-article-footnote-id1

[6] J. Shewchuk, "Triangle," 2005. [Online]. Available: http://www.cs.cmu.edu/~quake/triangle.html

[7] C. Reynolds, *Steering Behaviours For Autonomous Characters*, 1985. [Online]. Available: http://www.red3d.com/cwr/papers/1999/gdc99steer.pdf

[8] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[9] R. Belwariar, "A* search algorithm - geeksforgeeks." [Online]. Available: https://www.geeksforgeeks.org/a-search-algorithm/

[10] C. Simpson, "Behavior trees for ai: How they work," 2014. [Online]. Available: http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

[11] R. Monteiro, "Understanding the game main loop — higher-order fun," 2010. [Online]. Available: http://higherorderfun.com/blog/2010/08/17/understanding-the-game-main-loop/

[12] K. Witters, "dewitters game loop – dewitters." [Online]. Available: https://dewitters.com/dewitters-gameloop/

[13] "performance comparison of json libraries: jackson vs gson vs fastjson vs json.simple vs jsonp_2018," 2018. [Online]. Available: https://interviewbubble.com/performance-comparison-of-json-libraries-jackson-vs-gson-vs-fastjson-vs-json-simple-vs-jsonp/

List of Figures

List of Tables

Listings