Team 8 COMP 477 Project Documentation

Maude Braunstein
27545967
maude.braunstein@gmail.com

Keven Presseau-St-Laurent
40000501
k_presse@encs.concordia.ca

Alexia Soucy
40014822
a_oucy@live.concordia.ca

Abstract

This report outlines the design decisions and relating to an OpenGL project implementing pathfinding in a static environment with dynamic obstacles in the form of morphing 4D geometry. The implementation makes use of 18-DOP bounding volumes.

1 Introduction

Team 8's project for COMP 477 (fall 2019) implements A* pathfinding across a plane with 4D obstacles that morph on user input by shifting the perspective along the W axis. A set of obstacles is randomly generated across the plane on launch along with a random start position and destination as well as random static obstacles in the environment.

1.1 Motivation and inspirations

The purpose of this project was to demonstrate an implementation for pathfinding that responds to a dynamic and unpredictably shifting environment as well as to develop a systematic way to render and control 4D geometry in a conventional 3D software environment.

1.1.1 4D

The 4D aspects of this application would have been much more difficult to implement without the existence of an online community of developers interested in 4D. The application's 4D aspects were originally inspired by the work of Mark ten Bosch and Jeff Weber on Miegakure[1] and the basic framework for implementing and controlling 4D geometry was partly inspired by a coding challenge created by Daniel Shiffman for his Coding Train YouTube channel wherein he displayed the 3D projection of a 4D hypercube. [2] Additionally, the 4D objects' 18-DOP bounding volume implementation was inspired by an 8-DOP implementation found in Christer Ericson's *Real-Time Collision Detection*. [3] The interest in implementing 4D geometry is in large part due to its behaviour when displayed in 3D, as it behaves as a morphing 3D object with no morphing code required.

1.2 Dependencies

This project requires the GLUT, GLM, and SOIL libraries, as well as the C++ Standard Template Library for some of its data structures. GLM is used for additional data structures, namely its mat and vec classes. The project also uses a precompiled header to handle its dependencies.

2 Methodology

This application's design is split among a first group of classes to handle the 4D obstacles and a second group to handle the pathfinding. A particle system is made to follow the path and leave a trail where it went.

2.1 Obstacles

The obstacles are managed through a set of three header files named dop18.h, gl4d.h, and obstacles.h with no cpp files. The ObstacleSet class in obstacles.h is used to manage a set of Hyperobjects as defined in gl4d.h. The Hyperobject class makes use of the DOP18 struct in dop18.h as well as its functions for collision detection. This approach was adopted to make it as easy as possible for teammates not tasked with 4D development to import and use it.

2.1.1 4D

The 4D components of the project are imported through gl4d.h, a header file declaring the Hyperobject class as well as its inheriting class Hypercube. Hypercube describes a 4D abstraction of a cube with normalized vertices along 4 dimensions.

The general definition of a hyperobject within this project's scope is a 3D object extruded into the 4th dimension along the W axis. This creates a 4D shape with the same 3D object on either end and all their analogous faces linked by 4D prisms. A 4D hypercube, for instance, is made up of one cube with all its vertices positioned at -1 along the W axis, and another with all its vertices at 1 along the W axis. Each of their 6 faces are connected vertex-to-vertex to their 4D analog, forming another cube stretched along the 4th dimension and static along one other axis. For example, a cube face lying in the X and Y axes is linked to its 4D analog forming a cube along the X, Y, and W axes, with all vertices having the same value in Z. This implies that a hypercube is actually made up of 8 cubes. Based on this definition, in theory any model could be made four dimensional, but hypercubes are used in this implementation to demonstrate the principles at play.

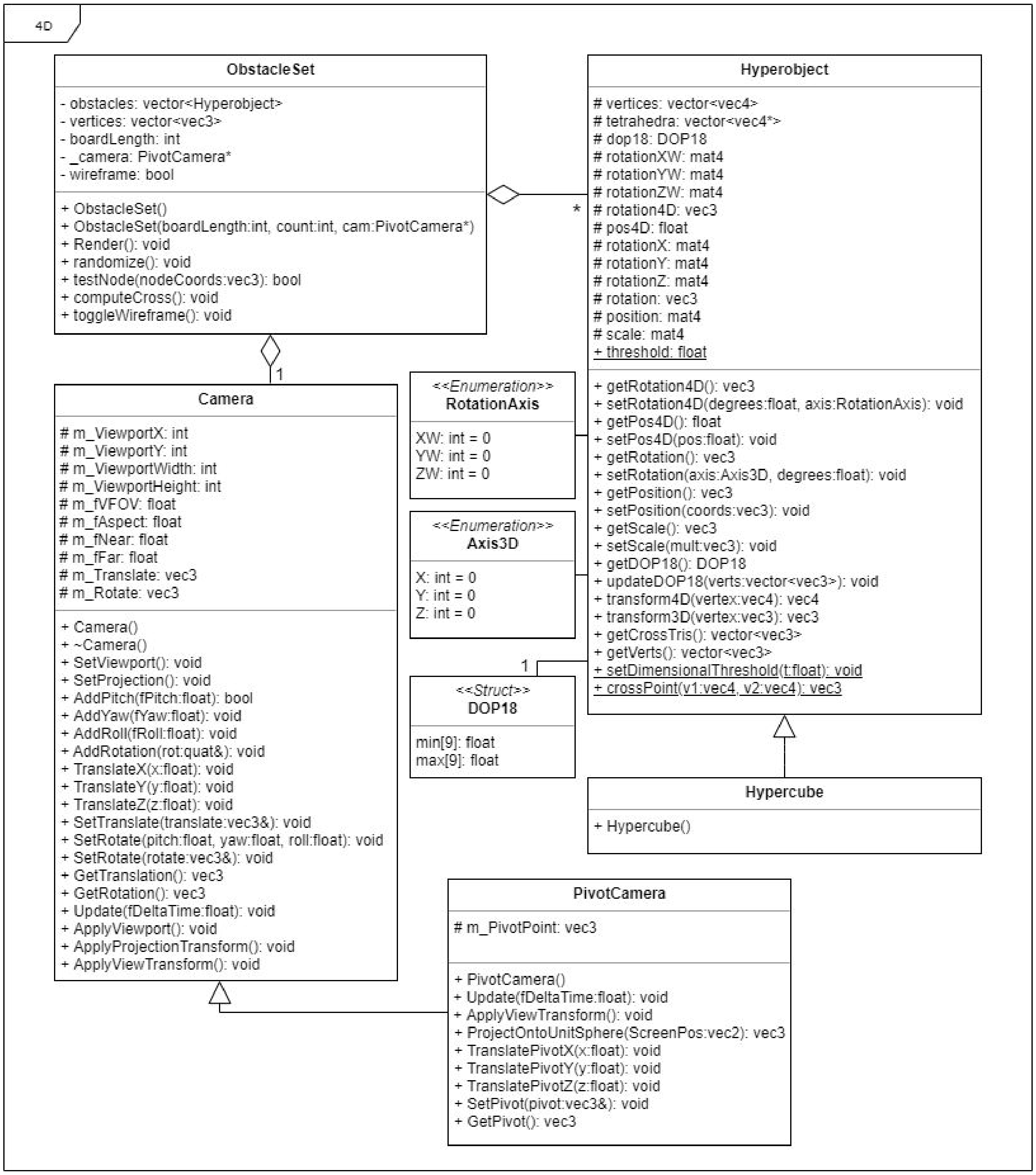
4D
ObstacleSet
- obstacles: vector<Hyperobject>
- vertices: vector<vec3>
- boardLength: int
- _camera: PivotCamera*
- wireframe: bool
+ ObstacleSet()
+ ObstacleSet(boardLength:int, count:int, cam:PivotCamera*)
+ Render(): void
+ randomize(): void
+ testNode(nodeCoords:vec3): bool
+ computeCross(): void
+ toggleWireframe(): void
*
1
Hyperobject
# vertices: vector<vec4>
# tetrahedra: vector<vec4*>
# dop18: DOP18
# rotationXW: mat4
# rotationYW: mat4
# rotationZW: mat4
# rotation4D: vec3
# pos4D: float
# rotationX: mat4
# rotationY: mat4
# rotationZ: mat4
# rotation: vec3
# position: mat4
# scale: mat4
+ threshold: float
+ getRotation4D(): vec3
+ setRotation4D(degrees:float, axis:RotationAxis): void
+ getPos4D(): float
+ setPos4D(pos:float): void
+ getRotation(): vec3
+ setRotation(axis:Axis3D, degrees:float): void
+ getPosition(): vec3
+ setPosition(coords:vec3): void
+ getScale(): vec3
+ setScale(mult:vec3): void
+ getDOP18(): DOP18
+ updateDOP18(verts:vector<vec3>): void
+ transform4D(vertex:vec4): vec4
+ transform3D(vertex:vec3): vec3
+ getCrossTris(): vector<vec3>
+ getVerts(): vector<vec3>
+ setDimensionalThreshold(t:float): void
+ crossPoint(v1:vec4, v2:vec4): vec3
Camera
# m_ViewportX: int
# m_ViewportY: int
# m_ViewportWidth: int
# m_ViewportHeight: int
# m_fVFOV: float
# m_fAspect: float
# m_fNear: float
# m_fFar: float
# m_Translate: vec3
# m_Rotate: vec3
+ Camera()
+ ~Camera()
+ SetViewport(): void
+ SetProjection(): void
+ AddPitch(fPitch:float): bool
+ AddYaw(fYaw:float): void
+ AddRoll(fRoll:float): void
+ AddRotation(rot:quat&): void
+ TranslateX(x:float): void
+ TranslateY(y:float): void
+ TranslateZ(z:float): void
+ SetTranslate(translate:vec3&): void
+ SetRotate(pitch:float, yaw:float, roll:float): void
+ SetRotate(rotate:vec3&): void
+ GetTranslation(): vec3
+ GetRotation(): vec3
+ Update(fDeltaTime:float): void
+ ApplyViewport(): void
+ ApplyProjectionTransform(): void
+ ApplyViewTransform(): void
<<Enumeration>>
RotationAxis
XW: int = 0
YW: int = 0
ZW: int = 0
<<Enumeration>>
Axis3D
X: int = 0
Y: int = 0
Z: int = 0
1
<<Struct>>
DOP18
min[9]: float
max[9]: float
Hypercube
+ Hypercube()
PivotCamera
# m_PivotPoint: vec3
+ PivotCamera()
+ Update(fDeltaTime:float): void
+ ApplyViewTransform(): void
+ ProjectOntoUnitSphere(ScreenPos:vec2): vec3
+ TranslatePivotX(x:float): void
+ TranslatePivotY(y:float): void
+ TranslatePivotZ(z:float): void
+ SetPivot(pivot:vec3&): void
+ GetPivot(): vec3

Fig. 1. 4D functionality class diagram

2.1.1.1 Technical Considerations

Some technical considerations must be made to properly generalize geometric concepts in 4D and then compute their 3D equivalent dynamically. Most notably, the 4D mesh must be constructed as a volume of tetrahedra rather than a surface of triangles.

This approach is necessary to properly turn the 3D cross-section of a set of 4D vertices into a triangle mesh. A tetrahedron's cross-section is always either a triangle or a quadrilateral that can be defined as two triangles. On the other hand, a triangle's cross-section with a plane is always a line segment. A 4D mesh of tetrahedra, then, will automatically generate a 3D mesh of triangles when its cross-section is computed. A 4D mesh of triangles, on the other hand, will generate a set of subdivided edges for which cannot be trivially turned into a triangle mesh.

2.1.1.2 Hyperobject

The Hyperobject class declares a set of member functions and data structures used to contain 4D geometry and generate a 3D cross-section of it for use in OpenGL. It allows for transformations in both 3D and 4D as well as 18-DOP bounding (through the dop18.h header file described in subsection 2.1.2).

static vec3 crossPoint(vec4 v1, vec v2)

Finds the point between two 4D vertices where the line segment they draw intersects with the dimensional threshold. Uses linear interpolation to find the intersection 3D coordinates.

vec3 getRotation4D()

Simple getter for a vector of rotation angles for the XW, YW, and ZW sets of axes. These combinations allow for 4D rotations using conventional rotation matrices.

void setRotation4D(float degrees, RotationAxis axis)

Simple setter for a 4D rotation about XW, YW, or ZW.

float getPos4D()

Simple getter for the Hyperobject's position along the W axis.

void setPos4D(float pos)

Simple setter for the Hyperobject's position along the W axis.

vec3 getRotation()

Simple getter for the rotation in degrees about all 3 conventional axes.

void setRotation(Axis3D axis, float degrees)

Simple setter for rotation about an axis.

vec3 getPosition()

Simple getter for the 3D position of the Hyperobjecct. Returns the meaningful parts of the positional matrix.

void setPosition(vec3 coords)

Simple setter for the 3D position of the Hyperobjecct. Applies the coordinates to the positional matrix.

vec3 getScale()

Simple getter for the 3D scale of the Hyperobject. Returns the meaningful parts of the scaling matrix.

void setScale(vec3 mult)

Simple setter for the 3D scale of the Hyperobject. Applies the multipliers to the scaling matrix.

DOP18 getDOP18()

Returns the Hyperobject's 18-DOP bounding volume.

void updateDOP18(vector<vec3> verts)

Updates the Hyperobject's 18-DOP bounding volume by checking if it must adjust its minima and maxima based on the passed vertices.

vec4 transform4D(vec4 vertex)

Applies 4D transforms to the Hyperobject's 4D vertices. This includes XW, YW, and ZW rotational combinations as well as the model's position along the W axis.

vec3 transform3D(vec3 vertex)

Applies 3D transforms to the Hyperobject's 3D cross-section vertices. This includes all conventional 3D transformations.

vector<vec3> getCrossTris()

Returns the cross-section of the Hyperobject by computing the triangles that result from tetrahedra intersecting with the plane described by the dimensional threshold. This creates a 3D model acting as the 4D Hyperobject's cross-section. The return vector has the vertices ordered to render as triangles.

vector<vec3> getVerts()

Returns the transformed 3D vertices of the Hyperobject's cross-section ordered for triangle rendering.

2.1.1.3 Hypercube

As described at the top of this section, a hypercube is a cube extruded into the 4th dimension, resulting in 2 cubes lying at -1 and 1 along the W axis with each 8 vertices normalized along the standard X, Y, and Z axes. The cubes' analogous faces are linked along the 4th dimension, resulting in a total of 8 cubes. Each cube is described as a mapping of 5 tetrahedra, for a total of 16 vertices and 40 tetrahedra.

Hypercube()

Basic constructor for a Hypercube. Fills the vertices vector with normalized hypercube vertex coordinates and maps the tetrahedra vector to the appropriate vertices to model a full 4D hypercube made of tetrahedra for use in OpenGL.

2.1.2 Collision Detection

To determine if a node on the board is obstructed by an obstacle, each 4D obstacle possesses an 18-DOP bounding volume as detailed in dop18.h. This header file defines a simple 18-DOP struct that keeps track of an object's extrema along 9 axes as follows: (1,0,0), (0,1,0), (0,0,1), (1,1,0), (-1,1,0), (0,1,1), (0,1,-1), (1,0,1), (1,0,-1) based on the implementation detailed in Ericson's *Real-Time Collision Detection*. [1] The extremas are contained in an array of minima (min[9]) and an array of maxima (max[9]). In addition to this, the 18-DOP can be checked either against another 18-DOP or a single vertex for intersections.

The decision to implement an 18-DOP was made based on the way hypercubes's cross-sections behave when rotated in 4 dimensions. Their corners have a tendency to bevel, which makes an 18-DOP with the above axes the most intuitive (and yet quite sufficiently light-weight for our purposes) approach. A simpler volume such as a bounding sphere, AABB, or OBB bounding volume would have posed challenges due to the high variety of shapes a 4D model's cross-section can take.

inline void setDOP18(vector<vec3> verts, DOP18& dop)

Computes the extrema for a model's 18-DOP bounding volume passed to it by checking the linear combination of each vertex along the 9 following axes: (1,0,0), (0,1,0), (0,0,1), (1,1,0), (-1,1,0), (0,1,1), (0,1,-1), (1,0,1), (1,0,-1).

inline void setDOP18(vec3 vert, DOP18& dop)

Initializes the 18-DOP extrema for one vertex to check collisions with other 18-DOP bounding volumes using the same axes.

inline bool testDOP18(DOP18 dop1, DOP18 dop2)

Tests for a collision between two 18-DOP bounding volumes by checking along every axis if either dop1's minimum is larger than dop2's maximum or dop1's maximum is smaller than dop2's minimum, in which case there is no intersection and false is returned. If none of the axes satisfy these requirements, true is returned to signal an intersection.

inline bool testDOP18(DOP18 dop, vec3 p)

Tests for a collision between an 18-DOP bounding volume and a single point to find if the volume occupies the space described by the point. This is used to find if a node on the pathfinding board is obstructed.

2.1.3 Obstacle Management

4D obstacles in this project are managed through the ObstacleSet class. This automatically generates a set of 4D obstacles with randomized orientations, positions, scales and 4D offsets, creating the impression of large boulders that can morph in 4D. ObstacleSet also implements a basic algorithm to check if a node on the pathfinding board is intersecting with any obstacle's 18-DOP bounding volume, obstructing the node. The ObstacleSet can render the Hyperobjects with a wireframe superimposed when the F key is pressed.

ObstacleSet()

Default constructor for ObstacleSet objects. Does not initialize any components.

ObstacleSet(int boardLength, int count, PivotCamera* cam)

Constructor for ObstacleSet objects that takes a board length and Hyperobject count to generate a collection of Hyperobjects to serve as obstacles. Takes a camera pointer for rendering.

void Render()

Renders the obstacles' 3D cross-sections with vertices colour-scaled based on their X, Y, and Z coordinates so their forms are easier to tell apart. If wireframe mode is on, this will also add a black wireframe on top of the geometry to display obstacle topology.

void randomize()

Randomizes each obstacle's position based on board length, orientation on every axis, and scale. The scale multiplier is between 2.5 and 5 on the X and Z axes and between 5 and 10 on the Y axis. Also randomizes the 4D offset between -2.5 and 2.5 so not all obstacles will appear at the same time.

bool testNode(vec3 nodeCoords)

Checks a pathfinding node's coordinates against the 18-DOP bounding volume of every obstacle to find if it is obstructed. If and intersection is found, returns true. Returns false otherwise.

void computeCross()

Computes the cross-section of each Hyperobject to store all together in the ObstacleSet's vertices collection. Ideally, this is only called when the user morphs the 4D objects by moving along the W axis. The vertices are ordered to render as triangles.

void toggleWireframe()

Toggles wireframe mode for use by Render().

2.2 Pathfinding

The principal idea behind the system implemented to produce the pathfinding for the AI was to create a world with multiple levels of subdivisions where the tiniest subdivisions would hold a node. These nodes were then all relayed by adjacency. This would create a graph layer over the world subdivision that describes the available paths. Afterwards we implemented A* with a simple heuristic to find an efficient path after reaching each node.

2.2.1 World Subdivisions

By order of size, the Board is a square that contains M x M Tiles. Each Tile is split into four square quadrants which are then split into two triangles.

2.2.1.1 Board

The Board class is the main class used to hold most of the methods and is used to avoid making calls to subsections from the main. It contains M x M Tiles of size (side length)/M. A Board holds tiles in the following order:

n+2	n+5	n+8
n+1	n+4	n+7
n	n+3	n+6

2.2.1.2 Tile

Tiles are separated into 4 quadrants as follows:

n+1	n+3
n	n+2

2.2.1.3 Quadrants

Quadrants defined by the SubSquare class are separated into two triangles as follows:

\	n
n+1	\

2.2.1.4 Triangle Halfs

The halfs defined by the SubTriangle class hold the nodes used to generate the graph. Each node is connected to up to 3 adjacent nodes held by the adjacent triangles. Each triangle half can either be of type ROAD, as in the AI considers it a walkable path or ELSE which encapsulates any type of terrain or object where the AI cannot path through. A ROAD half can also be considered Obstructed, where the value of the heuristic is changed to represent the dynamically blocked path. Here's an example:

[Q1] {t3}\{t2}	[Q1] {t2}	[Q3] {t7}\{t6}	[Q3] {t6}
[Q1] {t3}	[Q1] {t3}\{t2}	[Q3] {t7}	[Q3] {t7}\{t6}
[Q0] {t1}\{t0}	[Q0] {t0}	[Q2] {t5}\{t4}	[Q2] {t4}
[Q0] {t1}	[Q0] {t1}\{t0}	[Q2] {t5}	[Q2] {t5}\{t4}

We can see here that the triangle t0 is adjacent to t1, t3 and t5.

2.2.2 Node-Graph System

As mentioned previously, there is an overlaying Node-Graph system on top of the Board. The sole purpose of this system is to facilitate the execution of the pathfinding algorithm. By having a graph of connected nodes, it becomes easy to implement an algorithm such as Djikstra or A*. Since each node has up to 3 nodes they are connected it is simple to implement a container of pointers to those nodes.

2.2.3 A*

A* is one of the simpler yet really effective pathfinding algorithms, especially in a gaming context.
Our implementation of A*[4] consist of :

```
Setting the initial F(), G() and H() for each node
Creating an Open and Closed multiset
Insert the Player Node into Open
While Open is not empty:
    Pop the first Node (elements are ordered with a custom compare method in multisets)
    Add Node to Closed
    For each Connected Node:
        If Node is Target
            Stop
        Compute new G()
        If Node is in Open and newG() < oldG()
            Remove from Open
        If Node is in Closed and newG() < oldG()
            Remove from Closed
        If Node is not in Open nor Closed
            Set G() as new G()
            Set F() = G() + H()
            Add previous Node as Parent
            Insert Node in Open
Finally start from the Target Node and iterate through the parent Nodes to build the path
```

2.2.4 Heuristic

The Heuristic used is a pretty straightforward one to make backtracking a lot more costly. H() is defined as the distance from the target + distance from the AI + a weight if the node is further away from the target than the AI and finally a weight if the node is obstructed. This way, any nodes behind the AI is less likely to be travelled to, the same applies to nodes extending away from the flight path.

2.3 Particle System

The particle system is made up of 2 parts: The particle effect and the generator.[5]

2.3.1 Particle effect

The particle effect consists of a single particle with all its properties. It consists of a billboard particle which means it’s a 2D object always facing the camera. Each particle is given an initial speed and gravity. It’s speed is updated each update and each particle is updated during Idle cycles, but they are all pushed into a VertexBuffer and drawn at the same time to avoid drawing each particle separately.

2.3.2 Particle Generator

The generator consists of the shape from which particles can be generated. In this case, we used a cylindrical prism where each particle is spawned randomly within its boundaries. As its origin moves, the particles are offset so that once they are spawned they don't look attached to the generator.

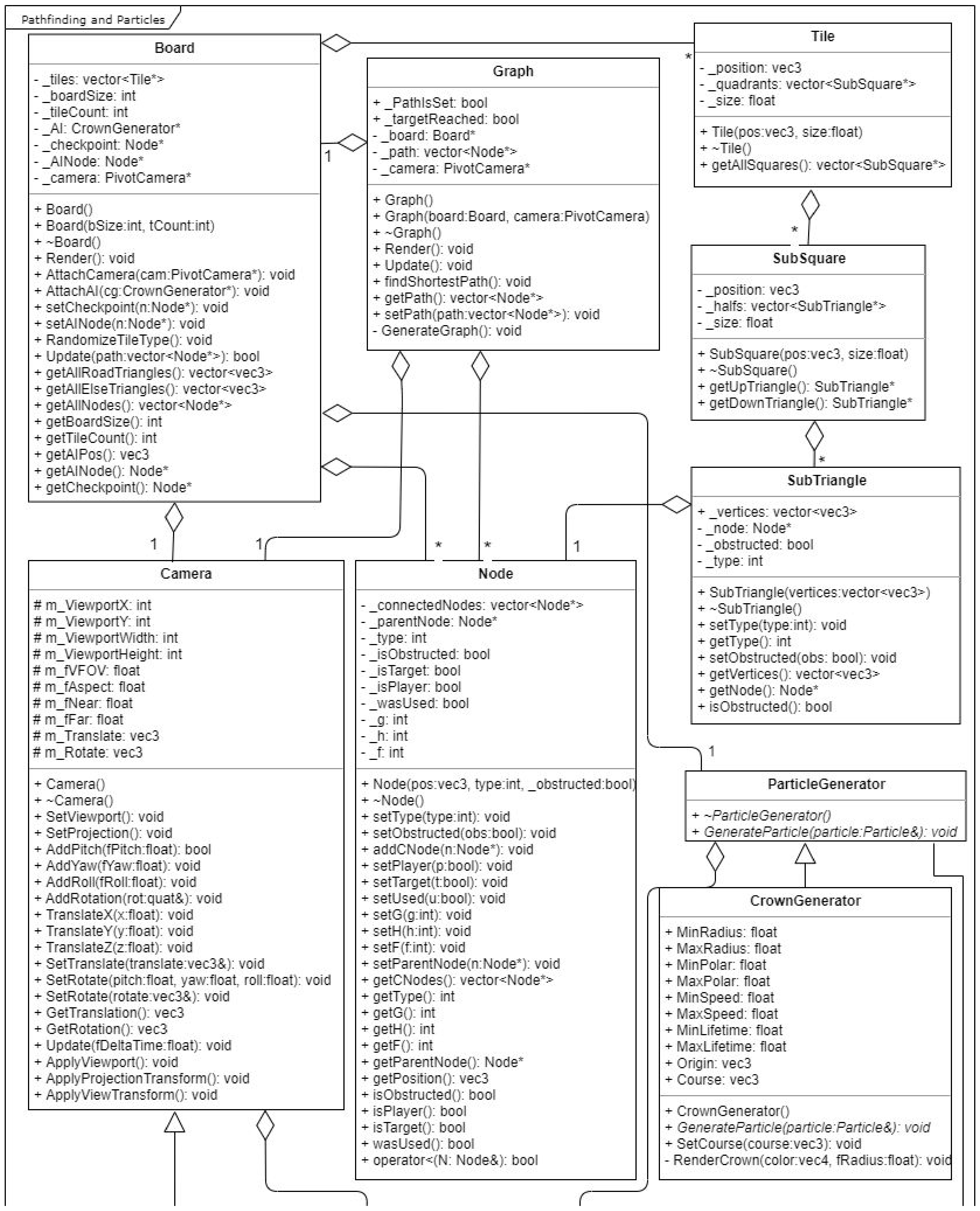
Pathfinding and Particles
Board
- _tiles: vector<Tile*>
- _boardSize: int
- _tileCount: int
- _AI: CrownGenerator*
- _checkpoint: Node*
- _AINode: Node*
- _camera: PivotCamera*
+ Board()
+ Board(bSize:int, tCount:int)
+ ~Board()
+ Render(): void
+ AttachCamera(cam:PivotCamera*): void
+ AttachAI(cg:CrownGenerator*): void
+ setCheckpoint(n:Node*): void
+ setAINode(n:Node*): void
+ RandomizeTileType(): void
+ Update(path:vector<Node*>): bool
+ getAllRoadTriangles(): vector<vec3>
+ getAllElseTriangles(): vector<vec3>
+ getAllNodes(): vector<Node*>
+ getBoardSize(): int
+ getTileCount(): int
+ getAIPos(): vec3
+ getAINode(): Node*
+ getCheckpoint(): Node*
Graph
+ _PathIsSet: bool
+ _targetReached: bool
- _board: Board*
- _path: vector<Node*>
- _camera: PivotCamera*
+ Graph()
+ Graph(board:Board, camera:PivotCamera)
+ ~Graph()
+ Render(): void
+ Update(): void
+ findShortestPath(): void
+ getPath(): vector<Node*>
+ setPath(path:vector<Node*>): void
- GenerateGraph(): void
Tile
- _position: vec3
- _quadrants: vector<SubSquare*>
- _size: float
+ Tile(pos:vec3, size:float)
+ ~Tile()
+ getAllSquares(): vector<SubSquare*>
SubSquare
- _position: vec3
- _halfs: vector<SubTriangle*>
- _size: float
+ SubSquare(pos:vec3, size:float)
+ ~SubSquare()
+ getUpTriangle(): SubTriangle*
+ getDownTriangle(): SubTriangle*
SubTriangle
+ _vertices: vector<vec3>
- _node: Node*
- _obstructed: bool
- _type: int
+ SubTriangle(vertices:vector<vec3>)
+ ~SubTriangle()
+ setType(type:int): void
+ getType(): int
+ setObstructed(obs: bool): void
+ getVertices(): vector<vec3>
+ getNode(): Node*
+ isObstructed(): bool
Camera
# m_ViewportX: int
# m_ViewportY: int
# m_ViewportWidth: int
# m_ViewportHeight: int
# m_fVFOV: float
# m_fAspect: float
# m_fNear: float
# m_fFar: float
# m_Translate: vec3
# m_Rotate: vec3
+ Camera()
+ ~Camera()
+ SetViewport(): void
+ SetProjection(): void
+ AddPitch(fPitch:float): bool
+ AddYaw(fYaw:float): void
+ AddRoll(fRoll:float): void
+ AddRotation(rot:quat&): void
+ TranslateX(x:float): void
+ TranslateY(y:float): void
+ TranslateZ(z:float): void
+ SetTranslate(translate:vec3&): void
+ SetRotate(pitch:float, yaw:float, roll:float): void
+ SetRotate(rotate:vec3&): void
+ GetTranslation(): vec3
+ GetRotation(): vec3
+ Update(fDeltaTime:float): void
+ ApplyViewport(): void
+ ApplyProjectionTransform(): void
+ ApplyViewTransform(): void
Node
- _connectedNodes: vector<Node*>
- _parentNode: Node*
- _type: int
- _isObstructed: bool
- _isTarget: bool
- _isPlayer: bool
- _wasUsed: bool
- _g: int
- _h: int
- _f: int
+ Node(pos:vec3, type:int, _obstructed:bool)
+ ~Node()
+ setType(type:int): void
+ setObstructed(obs:bool): void
+ addCNode(n:Node*): void
+ setPlayer(p:bool): void
+ setTarget(t:bool): void
+ setUsed(u:bool): void
+ setG(g:int): void
+ setH(h:int): void
+ setF(f:int): void
+ setParentNode(n:Node*): void
+ getCNodes(): vector<Node*>
+ getType(): int
+ getG(): int
+ getH(): int
+ getF(): int
+ getParentNode(): Node*
+ getPosition(): vec3
+ isObstructed(): bool
+ isPlayer(): bool
+ isTarget(): bool
+ wasUsed(): bool
+ operator<(N: Node&): bool
ParticleGenerator
+ ~ParticleGenerator()
+ GenerateParticle(particle:Particle&): void
CrownGenerator
+ MinRadius: float
+ MaxRadius: float
+ MinPolar: float
+ MaxPolar: float
+ MinSpeed: float
+ MaxSpeed: float
+ MinLifetime: float
+ MaxLifetime: float
+ Origin: vec3
+ Course: vec3
+ CrownGenerator()
+ GenerateParticle(particle:Particle&): void
+ SetCourse(course:vec3): void
- RenderCrown(color:vec4, fRadius:float): void
1
*
1
1
*
*
1
*
*
1

Fig. 2. Pathfinding and particle effect functionality class diagram Part 1

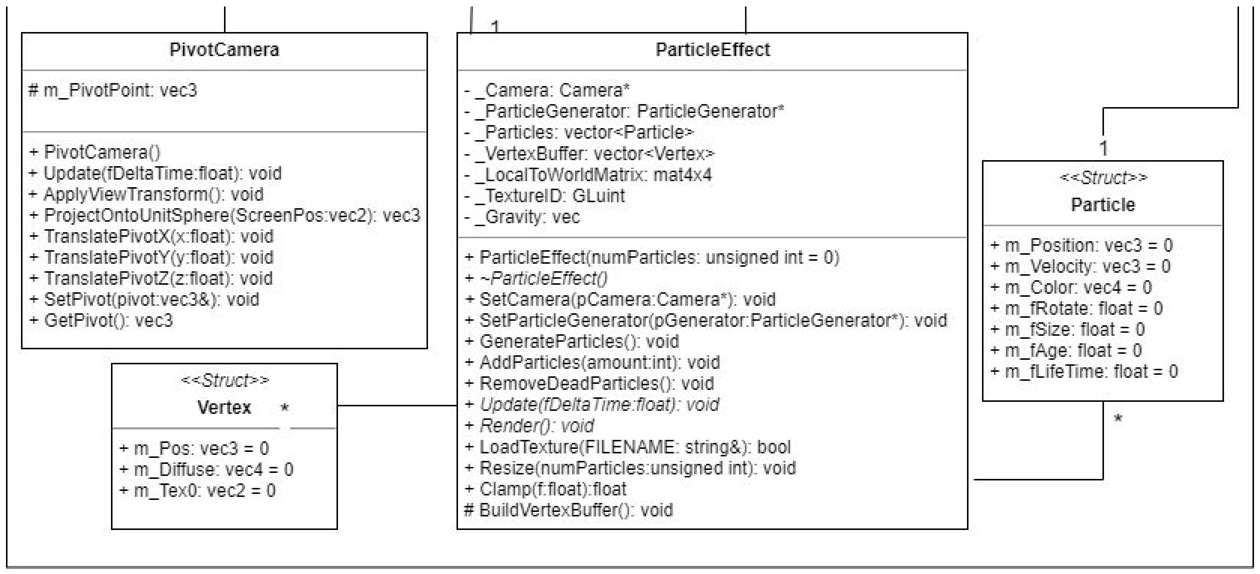
PivotCamera
# m_PivotPoint: vec3
+ PivotCamera()
+ Update(fDeltaTime:float): void
+ ApplyViewTransform(): void
+ ProjectOntoUnitSphere(ScreenPos:vec2): vec3
+ TranslatePivotX(x:float): void
+ TranslatePivotY(y:float): void
+ TranslatePivotZ(z:float): void
+ SetPivot(pivot:vec3&): void
+ GetPivot(): vec3
1
ParticleEffect
- _Camera: Camera*
- _ParticleGenerator: ParticleGenerator*
- _Particles: vector<Particle>
- _VertexBuffer: vector<Vertex>
- _LocalToWorldMatrix: mat4x4
- _TextureID: GLuint
- _Gravity: vec
+ ParticleEffect(numParticles: unsigned int = 0)
+ ~ParticleEffect()
+ SetCamera(pCamera:Camera*): void
+ SetParticleGenerator(pGenerator:ParticleGenerator*): void
+ GenerateParticles(): void
+ AddParticles(amount:int): void
+ RemoveDeadParticles(): void
+ Update(fDeltaTime:float): void
+ Render(): void
+ LoadTexture(FILENAME: string&): bool
+ Resize(numParticles:unsigned int): void
+ Clamp(f:float):float
# BuildVertexBuffer(): void
1
<<Struct>>
Particle
+ m_Position: vec3 = 0
+ m_Velocity: vec3 = 0
+ m_Color: vec4 = 0
+ m_fRotate: float = 0
+ m_fSize: float = 0
+ m_fAge: float = 0
+ m_fLifeTime: float = 0
*
<<Struct>>
Vertex
*
+ m_Pos: vec3 = 0
+ m_Diffuse: vec4 = 0
+ m_Tex0: vec2 = 0

Fig. 3. Pathfinding and particle effect functionality class diagram Part 2

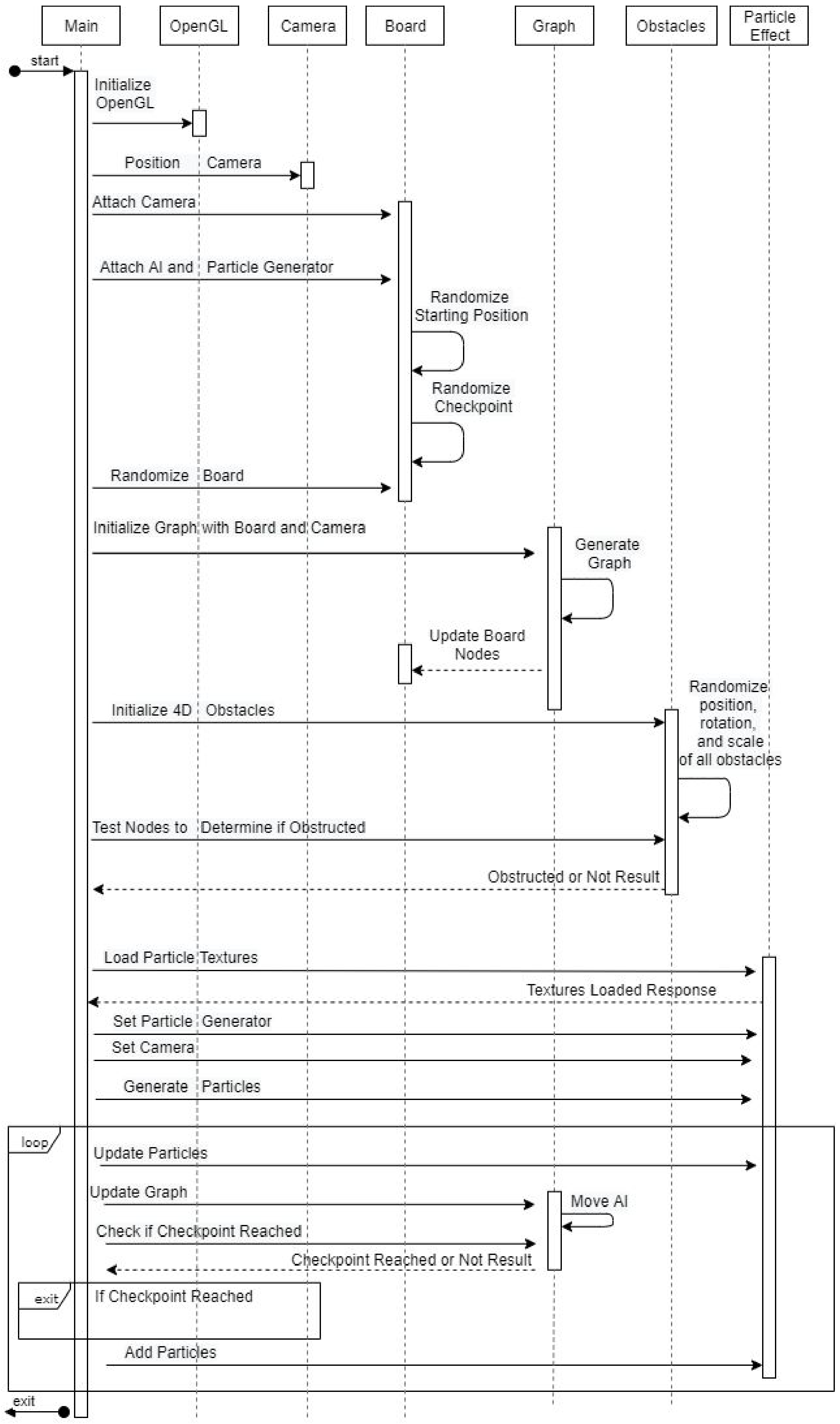

Fig. 4. Sequence diagram of the project

6 Conclusion

While the core features of the project are all presently implemented, some additional features that had been discussed at first had to be excluded for lack of time. Such features include the presence of a pursuing NPC that would chase the player character and possibly implement flocking behaviour when multiple were present as well as the ability to select a new destination through a bird's eye view camera. Additionally, the originally planned cityscape was altered to create a desert environment once it was realized that the 4D geometry had a tendency to look like crystals or boulders, making them more aesthetically fitting in a natural environment.

6.1 Challenges

6.1.1 4D

The current 4D implementation could be optimized and made more flexible given more time.

The current algorithms to generate a vector of cross-section vertices is quite costly and makes it so the 4D objects are difficult to use in large numbers, especially given that the more complex the base 3D form is, its 4D equivalent is exponentially complex. This has, however, been somewhat alleviated by making the ObstacleSet only recalculate the obstacles' cross-section on demand when the user makes the 4D objects morph.

The current approach of having preset 4D models as inheriting classes leaves little space for experimentation. Creating a new model out of tetrahedrons by hand can be time-consuming and requires a systematic approach. Ideally, it would be helpful to implement an algorithm to import 3D models and process them to subdivide into tetrahedra and extrude in 4D. Whether this is feasible, however, is outside the scope of this project.

In practice, the design decision of having the entire 4D code inside header files presented a challenge due to the pathfinding file hierarchy creating cyclic references when combined with the 4D headers. This was ultimately remedied, but having separate cpp files for the 4D code would have saved some trouble and not presented much additional overhead at compile time considering the 4D files are relatively small.

6.1.2 Pathfinding

A few of the shortcomings of this system stems from the fact that the base units are triangles. First off, using triangles ordered as they are right now instead of the standard rhombus inside a square pattern shifts all the paths towards a diagonal. Another shortcoming is the fact that we intuitively rotated around a triangle to connect its node to the adjacent nodes introducing a clockwise bias since the connected nodes were iterated through always using the same order. This introduced a problem such that the algorithm would always form paths heading in the same direction often creating less optimal solutions. Changing the order of the connected nodes and improving the heuristic corrected that bias pretty effectively.

6.2 Achievements

6.2.1 4D

The development of a fully functional class for 4D geometry was a challenge in itself, but finding a systematic way to structure 4D models as a set of tetrahedral mappings and getting to experiment with morphing geometry that completely circumvents the traditional morphing process has been quite rewarding.

6.2.2 Pathfinding

Implementing A* for the first time and defining the proper heuristic was definitely a challenge, but after drawing the results in the GL and seeing the path generated at each node it became a lot easier to finetune the algorithm. Implementing A* also required the usage of containers that are not commonly used such as Multiset which was fun to experiment with. However, the most complex part was to identify and access the proper adjacent triangles to connect the nodes together while they were stored in a linear vector of quadrants which were in a linear vector of tiles. Figuring out the right offsets was challenging, but by drawing them on the board they became easy to test and fix.

6.3 Future

Given more time, it would have been possible to create a full-fledged game based on the application and experiment with different 4D volumes and pathfinding algorithms.

Based on the experience gained developing the 4D components of this project, it would be interesting to explore avenues for automating the importing, 4D extrusion, and tetrahedral construction of user-defined models both to alleviate the process of defining 4D geometry in code but also to allow for more interesting geometry.

7 References

1. Ten Bosch, M., Weber, J. [Miegakure]. (2014, April 11). *Miegakure [Hide&Reveal] a true 4D puzzle-platforming game* [Video file]. Retrieved from https://www.youtube.com/watch?v=KhbUvoxjxIg
2. Shiffman, D. [Coding Train]. (2018, August 22). *Coding Challenge #113: 4D Hypercube (aka "Tesseract")* [Video file]. Retrieved from https://www.youtube.com/watch?v=XE3YDVdQSPo
3. Ericson, C. (2005) *Real-Time Collision Detection*. Retrieved from https://books.google.ca/books?id=WGpL6Sk9qNAC&pg=PA121#v=onepage&q&f=false
4. Implementation notes From Amit's Thoughts on Pathfinding. (n.d.). Retrieved December 19, 2019, from http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html.
5. Oosten, J. van. (2011, March 18). Simulating Particle Effects using OpenGL and C . Retrieved December 19, 2019, from https://www.3dgep.com/simulating-particle-effects-using-opengl/.