Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

Implementing Hexmap Generation Framework using Cube Coordinate System in Unity3D

Текстова частина до курсової роботи
за спеціальністю „ Інженерія програмного забезпечення ” 121

Керівник курсової роботи
к.т.н, доцент
Бублик В. В.

_____
(підпис)

“____” _____ 2021 р.

Виконав студент
Мартинюк Т.А

“____” _____ 2021 р.

Київ 2021
Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри інформатики,
проф., д.ф-м.н.
_____М. М. Глибовець
(підпис)
„____”_____2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту 4-го курсу, факультету інформатики
Мартинюку Тарасу Адамовичу

Розробити дизайн системи представлення шестикутникової карти для використання у програмуванні ігор

Вихідні дані:
- Дизайн системи представлення шестикутникової карти для гри;
- Імплементація Кубічної системи координат, ключових алгоритмів переходу від 3Д світу до цієї системи
- Система імпортування карти з програми-дизайнера Tiled
- Дослідження Кубічної системи координат та її аналогів

Зміст ТЧ до магістерської роботи:
Зміст
Анотація
Вступ
1. Дослідження Кубічної системи координат та її аналогів.
2. Імплементація потрібних алгоритмів в Кубічній системі координат.
3. Дизайн системи представлення шестикутникової карти для гри.
Висновки
Список літератури
Додатки

Дата видачі „___” _____ 2021 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Тема: Implementing Hexmap Generation Framework using Cube Coordinate System in Unity3D

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	02.11.2021	
2.	Огляд технічної літератури за темою роботи.	15.01.2021	
3.	Розділ 1	14.02.2021	
3.	Розділ 2	03.03. 2021	
4.	Розробити систему представлення шестикутникової карти для гри	01.04. 2021	
5.	Розділ 3	07.04. 2021	
7.	Написання пояснювальної роботи.	20.04.2021	
8.	Створення слайдів для доповіді та написання доповіді.	22.04.2021	
11.	Остаточне оформлення пояснювальної роботи та слайдів.	3.04.2021	
12.	Захист курсової роботи	15.04.2021	

Студент _____
Керівник _____
“_____” _____

Implementing Hexmap Generation Framework using Cube Coordinate System in Unity3D

Taras Martyniuk,
National University of Kyiv-Mohyla Academy
School of Information Technology

Kyiv 2021

Contents

Abstract

This paper describes the implementation of hexagonal grid framework that uses Cube coordinate system for hexagon representation and implementation of algorithms. Unity3D game engine was used, along with its Entity Component System framework for runtime grid representation and Tiled tilemap editor for level design.

It also showcases and compares different coordinate systems used for representing hexagonal grids.

Keywords: Game Development, Hexagonal Grids, Hexmap,

Introduction

This document combines the best practices of working with hexagonal grids into a single framework, available to the users of Unity3D – one of the two most popular non-proprietary game engines in the world.

The first chapter of this study explains the need for grids in computer games' world representation and overviews different techniques for it. The second analyzes the two main approaches for defining a coordinate system for hexagonal grids. Then, the third chapter proposes my implementation of a gameplay-level framework for working with hexagonal grids using Cube coordinate system, including generating the map from data exported from external editor, implementing the math for coordinate systems, representing the map on runtime, and implementing example pathfinding algorithms on it.

1. Usage of Hexagonal Grids in Computer Games

The main goal of using grids in computer games is to divide a continuous 3D world space, in which every point can have arbitrary precision, to a more constrained system of units with lower precision. This is important to abstract away unnecessary parts of the world – such as the real number of units (e.g., meters) some area may have, and operate on a higher level. This is needed for both players and designers. It allows for more convenient reasoning of the world and its gameplay properties, as well as simplifies some computations – mainly pathfinding, by effectively dramatically lowering the precision of the processed data.

The most common grid form in games is square – a subdivision of some 2D plane in the game world into a grid where each tile spans N real units of game world. They are most popular in games with turn-based combat, such as some from the Heroes of Might and Magic or XCOM series. These games have grids at their core representation of the world, and clearly show that structure to users. However, even games that don't do that may still use grids under the hood, mainly for pathfinding purposes. Real time strategy games – e.g., Warcraft and StarCraft – are an example of this.

Figure 1 - square grids in Heroes of Might and Magic 7

While being heavily used, square grids do lack some features. The main issue with them is a problem with directional movement – diagonal distances distort grid movement, since while we move only one hex, we actually traverse a larger amount in-world.

Hexagonal grids offer a solution for this problem. They distort the distances less, mainly because of the fact that each hexagonal tile has more non-diagonal neighbors than a square one. Most prominent examples of hexagonal grids are found in 4X games, like Civilization series, as well as tabletop games.

Figure 2 - Hexagonal map in Civilization 6

2. Overview of coordinate systems for Hex Grids

2.1. Offset Coordinate System

Disregarding the coordinate system, every hexagon grid can have two variations: hexagons can be aligned with either flat top or pointy top.

The most common way to represent hex grids is to re-use the general 2D cartesian system from the square grids – with the axis aligned with columns or rows. This is known as Offset coordinate system. With Offset system another variation comes into play - You can either offset the odd or the even column/rows.

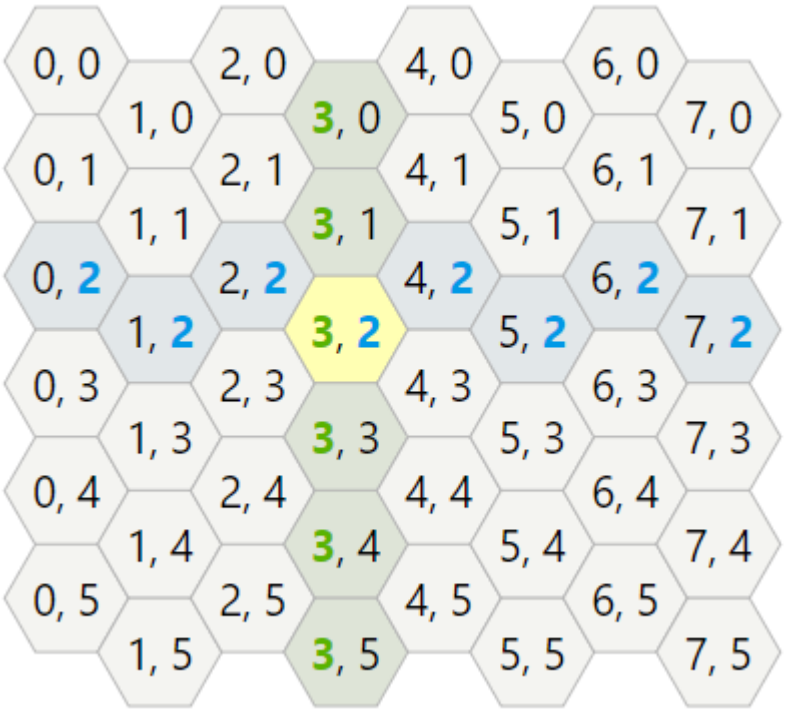


Figure 3Offset coordinate system for flat-top odd-column layout

2.2. Cube Coordinate System

Cube coordinate system introduces a third axis – and basis. The three resulting axes are aligned with the 3 pairs of opposite sides of a hexagon.

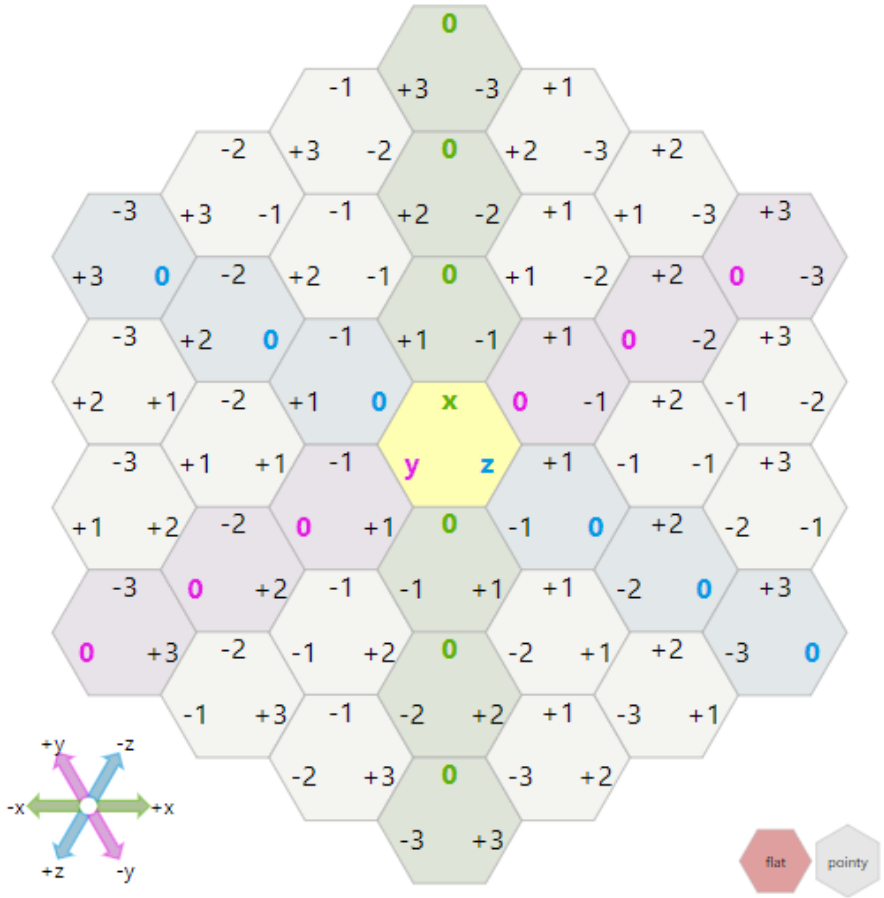


Figure 4 Cube coordinate system

2.3. Comparison

The added axis of cube system allows for a natural and much easier transition to neighboring hexes – which is crucial for all algorithms. The more complex basis also makes it possible to translate precisely from the outside coordinate system to the cube one, as we will see in next chapter.

Thus, the ideal way of working with hex grids would be to store the core functionality as cube, and only use offset for displaying to user, or in case of rectangular grids, for easier iteration over rows/columns (e.g. when generating world). While cube coordinates are generally best suited for most algorithms, offset coordinates are simpler to reason for a player.

3. Cube Coordinate system algorithms

As discussed in Chapter 2, hex grids can have different variations. For the sake of brevity, we will consider only the flat-top odd-row variant.

3.1. Conversion

Before going into the core operations over the coordinates which are done completely in the bounds of cube coordinate system, it is useful to explore how to convert points from outside coordinate systems to the cube one.

Cube system is constrained to a single plane. In 3D game, this would be the “terrain” plane, which designates the 0-height level of our world. In 2D games this plane will actually coincide with the single plane of our world. However, in both cases, the World Space is usually more precise than the Cube space. For cube space, 1 unit is a single hexagon, that in turn can span for an arbitrary amount of world space units. While most of our algorithms operate on data in Cube coordinate system (or even offset) since they allow for a simpler model of the world that is easier to understand and reason, at some point the results of these algorithms will have to be converted into the outside, 3D, world space coordinate system to be used by rendering systems and outputted to the screen.

Even in parts of our application that do not need to adhere to the external 3D world space, we may want to use both the Cube and offset coordinates for the reasons described in the Chapter 2.

3.2. World to Hex

Converting to and from World/Pixel coordinate systems is one of the operations that is much easier in the cube system than in the offset.

For cube coordinates, we can solve this task by using basis vectors. Note that we need only 2 basis vectors (of the Axial system), since the third can be derived from them – much
like the third coordinate of the Axial system.

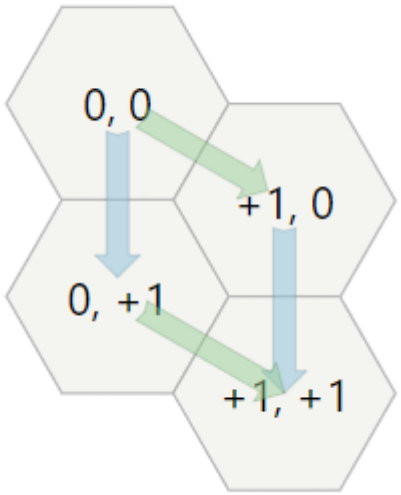


Figure 5- basis vectors of the Axial system [1]

To find a position of a hex in the world, we can then project the magnitudes of all it's coordinates onto this basis vectors, while correcting for the difference of unit lengths between Cube and World systems. This can be done efficiently as a

single matrix multiplication:

$$\begin{bmatrix} x \\ y \end{bmatrix} = size \times \begin{bmatrix} 3/2 & 0 \\ \sqrt{3}/2 & \sqrt{3} \end{bmatrix} \times \begin{bmatrix} q \\ r \end{bmatrix}$$

Figure 6 - transformation matrix for the conversion [1]

For the offset system there is no adequate way of achieving this, since its basis is too simplified – having only 2 axes that define the square grid. When working with offset coordinates the best way is to convert them into cube when this operation is required

3.3. Hex to World

The inverse conversion is done analogously, except that it requires an extra step that makes it more complicated.

The firs step is to do an inverse operation to the one we discussed in previous chapter – multiply the world/pixel coordinates with an inverse transformation matrix

$$\begin{bmatrix} q \\ r \end{bmatrix} = \begin{bmatrix} 2/3 & 0 \\ -1/3 & \sqrt{3}/3 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \div size$$

Figure 7 - Inverse transformation matrix [1]

```
function cube_round(cube):
    var rx = round(cube.x)
    var ry = round(cube.y)
    var rz = round(cube.z)

    var x_diff = abs(rx - cube.x)
    var y_diff = abs(ry - cube.y)
    var z_diff = abs(rz - cube.z)

    if x_diff > y_diff and x_diff > z_diff:
```

```
    rx = -ry-rz
  else if y_diff > z_diff:
    ry = -rx-rz
  else:
    rz = -rx-ry

  return Cube(rx, ry, rz)
```

Figure 6 - Pseudo-code for rounding fractional cube coordinate[1]

3.4. Pathfinding

Hex grid provides a natural basis for a graph. In fact, one of the main reasons for introducing grids into games is to have a more discrete and clear way of defining paths.

Every hex can be viewed as the node, connected to all the adjacent hexes. Most pathfinding algorithms require a way to get all those connected hexes for a given hex. In Cube system this is done similarly to any square grid – by adding all the basis vectors, with both signs, to the given hex.

3.5. Other Algorithms

Other algorithms known for cube system include: Distances, Rotation, Reflection, Rings, Field of View and are described by Amit Patel [1].

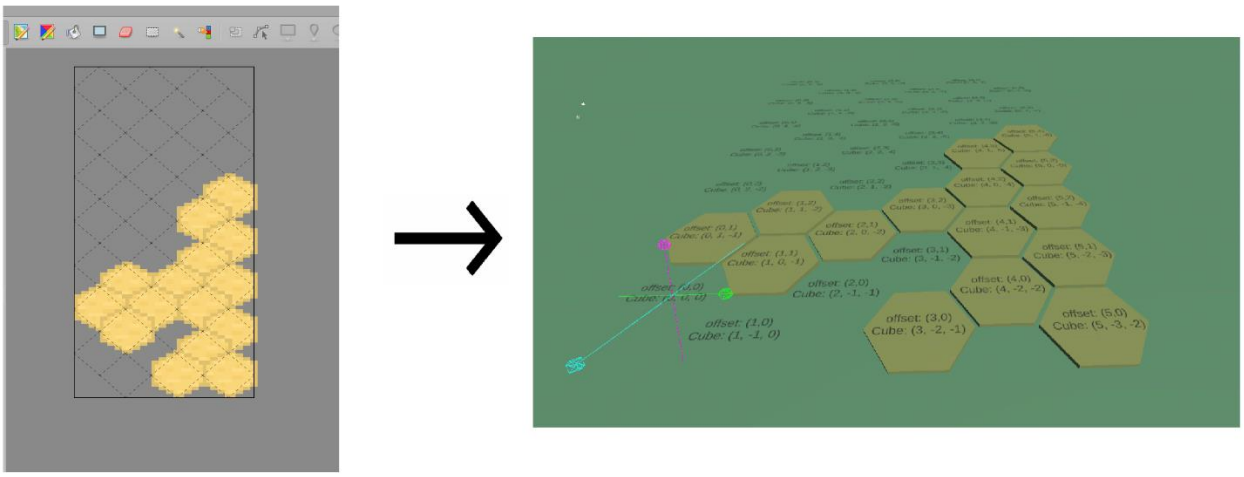
4. Implementing Hex Generation Framework in Unity3D

4.1. Tools and Framework Used

Unity3D is used as a game engine. It is a is a cross-platform game engine developed by Unity Technologies that uses C# as a scripting language [7]. It provides the game scene editor, runtime editor and basic entity architecture for this project.

The hexmap module is very closely tied to level-design, which is the part of game-development that deals with creating a lot of complex configuration that defines the whole world of the game – which needs to be tweaked manually while adding personal details (unless it is procedurally generated). Because of this, my system provides a workflow that is well suited for level-designers. The original hex maps are created in the Tiled tilemap editor [6]. It is a free and open-source GUI editor that allows creating tilemaps of different forms, including hexmaps that will be used in our project. Tiled provides an option to export the generated maps to Json format, which is then parsed by the tools provided in my framework to be

recreated in the Unity world.



8 - The hexmap created in Tiled(left) and recreated on runtime in Unity (right)

4.2. Usage Flow and Features

4.2.1. Prerequisites

A Hex map json file, created in Tiled hexmap editor and exported using the "export to json" feature.

4.2.2. Usage Flow and Setup

The setup of the system is done via the HexMapConfig unity component: [Screen]. After adding it to the scene users can set its configuration – set the path to the hexmap exported from tiled (`JsonMapFilename`), and a prefab Unity GameObject for every unique tile defined in the hexmap. This is how the users customize both

the visuals and data of each hex – for visuals they can edit the GameObject itself, e.g adding custom meshes and scripts manipulating them, and for data they can add the “GeneratingComponents” that will be converted into the ECS components and added to the corresponded Entity for this hex after conversion.

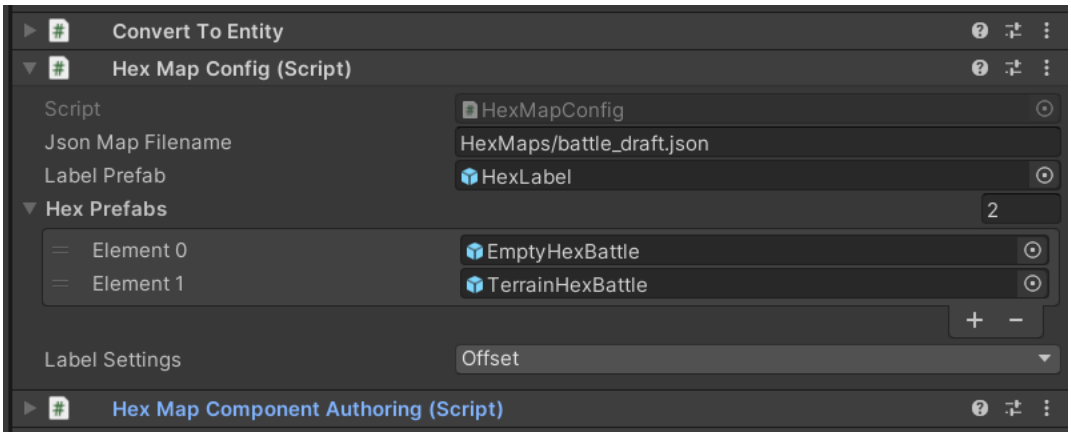


Figure 9 – Configuration UI for the HexGenerationSystem

When the game starts, this config is processed by our systems, and the following results are produced:

A hex lookup data-structure is created, accessible globally via ECS – this is the central way of accessing and storing any per-hex data of the game.

For every hex in the hexmap, a unique GameObject representing its visuals is instantiated as a copy of user-provided prefab and placed to a correct world-space position.

During each such prefab instantiation, an Entity holding that hexes’ data is created and added to the lookup. Users can customize the data of the Entity by adding AuthoringComponents in the editor. For any such component, a ECS Component will be added to Entity, with its initial data set by User in the editor.

4.2.3. Features

The main feature provided is the hex lookup itself, which can now be used for any gameplay task that requires information about the hex world and is automatically set up in the required configuration set in Tiled and Unity editor.

Additional features include implementation of Cube coordinate system – and its conversions to the Offset system, as well as selected algorithms for hex manipulation using Cube system – these are available as static utilities and HexNavigationSystem – a ECS system implementing pathfinding algorithms on Cube system - BFS search and retrieval of all hexes in range.

4.3. Architecture

The runtime data for the hexmap and related logic is designed in a Data-Oriented way instead of being Object-Oriented.

Data-oriented design was chosen because of two reasons. Firstly, from design point of view, it promotes the composition-over-inheritance mindset, leading to more scalable systems with fewer unnecessary abstractions. This is especially important since hexmap is a core module of every game, defining the whole game world structure, and is meant to be extended by users by adding their custom pieces of data potentially to any single hex without any constrains on the format/structure of the data, and meant to be used in unconstrained combinations.

Secondly, from the performance point of view, Data-Oriented design follows the “performance by default” mentality – meaning that by following its design guidelines users are bound to produce a program whose data-to-logic structure is well suited for the hardware and allows using numerous optimization that it

provides. Data layout in Data-Oriented layouts can be used much more easily by multithreaded algorithms since it minimizes the dependencies and allows for parallelization in more places. They are “cache-friendly” – designed with the processor cache usage workflow in mind, reducing the times processor needs to invalidate the cache when working with the data. This is important since hexmap contains data that is likely to have one of the largest scales in any project – since it defines the whole world map.

4.4. Implementation

4.4.1. Cube Coordinate System

Axial coordinates are used to represent the cube coordinates in the memory – since they provide a simple data representation. They are represented by `CubeCoords` struct [Appendix A]. Its interface completely hides the internal Axial system representation, since the third coordinate can always be generated. Offset coordinates are represented as a 2-dimensional vector (int2 struct provided by Unity3D).

All the math operations concerning the cube system are provided in the `HexMath` static utility class. Particularly, it implements the basic spacing calculations for a given radius (functions `HorizontalOffset`, `Height` and `Width`) [Appendix B].

The same `HexMath` class handles the conversion between Cube and Offset systems (`CubeToOffset` and `OffsetToCube` functions). Finally, it also implements the conversion to and from the world space. The 2D plane on which our hex grid is situated is defined as a `Plane` struct in HexUtils class. It is then used by HexMath -

function `PlanePointToHexCoord` converts a point on that plane to a cube coordinate system, and `GetHexCenter` does the opposite.

4.4.2. Map Generation and Storage

Generation happens in the HexGenerationSystem, which is a ECS system – the user input from editor needs to be retrieved by it by being tied to an entity that it queries for at the start of runtime. The `Generate` method does all the work – it parses the tilemap data from the json file, then maps the tile Ids to the user-provided prefabs. The prefabs are then converted to entities, while keeping the prefab itself – the generated entity is the instanced data-part of the tile, while `GameObject` is the presentation part. The global hex map is also configured with references to created entities.

The resulting hex map is stored as a buffer tied to a globally accessible entity. Its interface is different from its internal representation in memory – interface allows for a lookup of an Entity by a key of type int2, which is the offset coordinate for that hex. In actuality, the whole lookup is just a one-dimensional dynamic array of Entities. They are managed by Unity3D ECS, by defining the array as a IBufferElementData. This representation is the most memory and access efficient. To provide the described interface, however, users need to use a data-structure adapter – Slice2D. This is done for user in `GetCoordToHexArray` function – it retrieves the hexmap buffer and wraps it into the adapter:

```
var hexBuffer = entityManager.GetBuffer<HexEntityBufferElement>(hexOrigin.Value);

var hexMap = entityManager.GetComponentData<HexMapComponent>(hexOrigin.Value);
return new Slice2D<HexEntityBufferElement>(hexMap.Dimensions.y, new
NativeSlice<HexEntityBufferElement>(hexBuffer.AsNativeArray()));
```

Slice2D [Appendix C] is an adapter that wraps another adapter – NativeSlice and provides an interface that translates 2D indexing into “flat” indexing of NativeSlice and underlying DynamicBuffer.

Offset coordinates are used for the map storage, since they are more suitable for the square grids – using them for lookup results in no spare slots. This requires cube-offset-cube conversion on each lookup, but it is relatively cheap – running on constant time, and not requiring any heavy mathematical operations.

4.4.3. Algorithms

HexNavigationSystem [Appendix D] implements two pathfinding algorithms for our hex grid – `GetTilesInRange` and `GetPath`. Both use Breadth-First Search and a graph model of the hex grid. Graph Nodes are defined as individual hexes, plus extra data for the algorithms – `HexMoveGraphNode` and `HexMoveGraphPriorityNode` structures. Extra gameplay data for these algorithms is retrieved from the hex entities themselves – such as information about their movement cost. Users can freely extend it, as well as the pathfinding by inserting behavior that uses that data.

`GetTilesInRange` uses a simple BFS with array-based Queues, to find all the nodes reachable from start by N or less steps. The main advancement for the hex grid here is the way we compute neighboring nodes in our graph – this is done by simply adding and subtracting all the basis of our Cube system, much like we would do for a square grid or offset system. Several gameplay filters are provided for this algorithm.

`GetPath` implements a A* pathfinding algorithm on the equally defined graph of our hex grid. Instead of simple queue it uses a priority queue, allowing for a heuristic.

Conclusions

Hex grids are widely used for world representation and division into units, they are the most popular grid definition system. Currently used coordinate systems for representing hex grids are offset and cube.

Offset system is more straightforward and easier to reason and explain. Cube system, while having a more complex way of defining each hex, is more elegant and better fitted for hex grids, since it represents the laws (e.g directions and relative positioning) of hexagonal grids more naturally, which in turn simplifies most of the algorithms for operations defined on hexagonal grids.

Implementation of a framework for working with hexagonal grids was provided, showcasing the best uses for both Cube coordinate and offset systems, as well as proposing design for a lot of related problems, such as storing and looking up hexmap data, and making a gameplay framework extensible for the user.

References

[1] Amit Patel – Hexagonal Grids blog page https://www.redblobgames.com/grids/hexagons/

[2] **Charles Fu -** rec.games.programmer forum **-** http://www-cs-students.stanford.edu/~amitp/Articles/Hexagon2.html

[3] Hex Grids article on http://sc.tri-bit.com - Hex Grids - StoneHome (archive.org)

[4] **Clark Verbrugge** - article on hex grids - www.sable.mcgill.ca/~clump/hexes.txt

[5] Hex Grid Geometry article - http://playtechs.blogspot.com/2007/04/hex-grids.html

[6] Tiled Map Editor - https://www.mapeditor.org/

[7] Unity3D - https://unity.com/

Appendix A

Cube Coords

```
[Serializable]
public struct CubeCoords : IEquatable<CubeCoords>
{
    public int X;
    public int Z;
    // coordinate system constraint
    public int Y => -X - Z;
}
```

Appendix B

HexMath class

```
/// <summary>
/// using cube coordinate system, flat-top hexes, odd columns shoved by +1/2 of row
height
/// No offset for maps - HexMap origin must be at (0, 0)!
/// </summary>
public static class HexMath
{
    public static readonly float2x2 AxialBasis = new float2x2
    {
        c0 = new float2(3f / 2f, - math.sqrt(3) / 2f),
        c1 = new float2(0, - math.sqrt(3))
    };

    public static float HorizontalOffset(float radius)
    {
        return Width(radius) * 3 / 4;
    }

    public static float Width(float radius)
    {
        return 2 * radius;
    }

    public static float Height(float radius)
    {
        return math.sqrt(3) * radius;
    }

    public static int2 CubeToOffset(CubeCoords cube)
    {
        var col = cube.X;
        var row = cube.Z + (cube.X - (cube.X & 1)) / 2;
```

```
        // -row since our cube +Z is inverted with regards to what Amit uses
        return new int2(col, -row);
    }

    public static CubeCoords OffsetToCube(int2 offset)
    {
        var x = offset.x;
        // - y since our cube +Z is inverted with regards to what Amit uses
        var z = -offset.y - (offset.x - (offset.x & 1)) / 2;
        return new CubeCoords { X = x, Z = z };
    }

    /// <param name="point">on the hexmap z plane</param>
    public static CubeCoords PlanePointToHexCoord(float2 point, float hexRadius)
    {
        // x, z cube
        float2 fractCoords = math.mul(math.inverse(AxialBasis), point) / hexRadius;
        // fract coords are still subject to coord system constraint
        float fractCoordsY = -fractCoords.x - fractCoords.y;
        return CubeRound(new float3(fractCoords.x, fractCoordsY, fractCoords.y));
    }

    public static float2 GetHexCenter(CubeCoords coords, float hexRadius)
    {
        return math.mul(hexRadius * AxialBasis, new float2(coords.X, coords.Z));
    }

    /// <param name="fractCoords">Cube coords with float values</param>
    static CubeCoords CubeRound(float3 fractCoords)
    {
        float3 rounded = math.round(fractCoords);
        float3 diff = math.abs(rounded - fractCoords);

        if (diff.x > diff.y && diff.x > diff.z)
        {
            rounded.x = -rounded.y - rounded.z;
        }
        else if (diff.z > diff.y)
        {
            rounded.z = -rounded.x - rounded.y;
        }

        return new CubeCoords((int) rounded.x, (int) rounded.z);
    }

}
```

Appendix C

Slice2D

```
/// <summary>
/// adapter that interprets array(buffer) as 2D, without owning it
/// </summary>
public struct Slice2D<TElem> where TElem : struct
{
    public NativeSlice<TElem> FlatSlice => m_flatSlice;

    public int Size1D => m_flatSlice.Length / Size2D;
    public int Size2D { get;  set; }

    NativeSlice<TElem> m_flatSlice;


    public TElem this[int2 index]
    {
        get => m_flatSlice[FlattenIndex(index.x, index.y)];
        set => m_flatSlice[FlattenIndex(index.x, index.y)] = value;
    }

    public TElem this[int x, int y]
    {
        get => m_flatSlice[FlattenIndex(x, y)];
        set => m_flatSlice[FlattenIndex(x, y)] = value;
    }

    int FlattenIndex(int x, int y)
    {
        return x * Size2D + y;
    }
}
```

Appendix D

HexNavigationSystem algorithm

```
    // get all tiles in range from center, except those filtered out by hexFilter
    // if needsContinuousPath is true, each result tile has a Continuous path from center
    // BFS search
    public void GetTilesInRange(CubeCoords center, int range, GetTilesInRangeSettings
settings,
        Slice2D<HexEntityBufferElement> hexMap,
        List<HexMoveGraphNode> outMoves)
    {
        if (!hexMap.IsValidIndex(HexMath.CubeToOffset(center)))
        {
            Debug.LogError("Invalid center");
            return;
        }
        var visited = outMoves;
        visited.Clear();
        m_toVisit.Clear();

        m_toVisit.Enqueue(new HexMoveGraphNode(center, 0));

        // every iter is O(N) for searching in visited + amortized O(N) for que/deque
(array can shift)
        while (m_toVisit.Count != 0)
        {
            (CubeCoords current, int stepCount) = m_toVisit.Dequeue();

            int index = visited.FindIndex((HexMoveGraphNode node) => node.Coords ==
current);
            if (index != -1 && visited[index].StepCount <= stepCount)
            {
                continue;
            }

            var currentHex = hexMap[HexMath.CubeToOffset(current)].Hex;
            bool passesFilter = Filter(currentHex, settings);

            if (stepCount == range)
            {
                if (passesFilter)
                {
                    VisitCurrent();
                }
                continue;
            }

            if (current == center ||
                passesFilter || !settings.NeedsContinuousPath)
            {
                GetNeighbors(current, m_neighbors);
                foreach (CubeCoords neighbor in m_neighbors)
```

```
                {
                    int2 neighborOffset = HexMath.CubeToOffset(neighbor);
                    if (!hexMap.IsValidIndex(neighborOffset))
                    {
                        continue;
                    }

                    var neighborStepCount = stepCount + 1;
                    Debug.Assert(neighborStepCount <= range);

                    m_toVisit.Enqueue(new HexMoveGraphNode(neighbor, neighborStepCount));
                }
            }

            if (passesFilter)
            {
                VisitCurrent();
            }

            void VisitCurrent()
            {
                visited.Add(new HexMoveGraphNode(current, stepCount));
                WriteStepsToLabel(hexMap[HexMath.CubeToOffset(current)].Hex,
EntityManager, stepCount);
            }
        }

        Debug.Assert(visited.Distinct().Count() == visited.Count);
}
```