DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Simulating 2D Game Physics using dynamic Navigationgraphs

Lars Hinnerk Grevsmühl

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Simulating 2D Game Physics using dynamic Navigationgraphs

Simulation von 2D Spielphysik mit dynamischen Navigationsgraphen

Author:	Lars Hinnerk Grevsmühl
Supervisor:	Prof. Dr. Gudrun Johanna Klinker
Advisor:	Daniel Dyrda
Submission Date:	15.03.2020

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15.03.2020 Lars Hinnerk Grevsmühl

Acknowledgments

I am very grateful for the opportunity this paper provided. The Forschungsgruppe Augmented Reality (FAR) under the lead of Prof. Dr. Gudrun Johanna Klinker enabled me to write about the topic of my choice. Especially my advisor Daniel Dyrda supported me in this effort and helped along every step that was essential in the creation of this paper.

Without the help of my good friends Marlene, Felix, Marcel and Mehmet the completion of this paper would have been impossible. You helped me by proof reading the text, debugging the code, repairing the latex documents and motivating me to keep pushing forward.

My heartfelt thanks to all of you.

Abstract

The approach outlined in the following introduces a new way to simulate the physics of 2D game characters that cannot jump. The focus of the paper is on exploring the possibility of representing characters as a set of two points on a graph. These points move along the edges and resolve collisions. This was tested by implementing a prototype and developing several features around it. While the prototype was functional for a multitude cases, there were several exceptions that could not be solved with the current set of rules. A graph-based method was able to move the characters and to resolve collisions, but it failed at handling all cases that needed characters to be stacked on-top of each other. A graph-based approach is problematic when characters have a width, collide with each other and need to fall down ledges. If a game requires all three of these, it is recommended to use traditional collisions instead of the graph-based approach. If the game does not e.g. require the characters to collide with each other a graph-based solution might be beneficial.

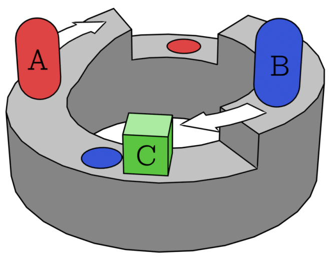
Contents

1 Introduction

1.1 Inspiration

The graph-based simulation approach presented in this paper was built to deliver a more predictable and easy way to simulate certain 2D puzzle games. The conceptual theory was tested with a prototype. Note that many concepts require more time and testing, only the underlying base was in scope for this paper. While some parts of the system were built to approximate real world physics, the main goal was to create a base for a semi realistic game simulation. Many of the choices were made to fit the game *Huge and Cute*, but the concept was designed to extend well to other games.

Huge and Cute is a game in which one player controls two characters on a cylindrical level as shown in Figure 1.2. Pushing the left key compels both characters to move clockwise, while the right key moves them counterclockwise. They cannot jump or perform any other actions. The goal for every level is to reach two buttons, which start the next level. Both characters always move at a constant speed unless one bumps into a wall, in which case that character stops while the other can keep moving. To increase the puzzle's complexity they each have different abilities (e.g. one can push boxes). They can interact with the physical elements in their environment by pushing objects or standing on them.



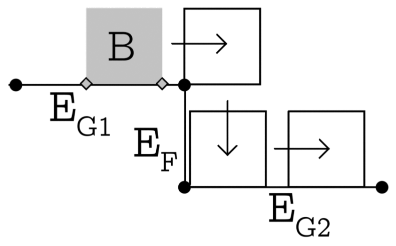
Characters A and B move towards their buttons and B pushes Box C

Figure 1.1: Huge and Cute movement

1.2 Target Specification

The ultimate goal is to design and implement a system that uses Navigation Graphs to simulate these characters efficiently and predictably. The standard approach to simulate these kinds of characters would be to use Rigidbodies. The challenges of this approach lie either either in their complexity and inefficiency (3D Rigidbody approach) or to unstable (wrapping 2D Rigidbodies around a cylinder). Due to these hindrances the focus will largely be on ensuring easy movement control for puzzle games.

Graphbodies represent all moving objects, such as characters or boxes. They will be represented by points on the edges of the Navigation Graph. They mimic the behavior of Rigidbodies with ground aligned box colliders. The navigation graph contains three types of edges. Besides jump edges, which will be introduced in Section 4.4 it contains *ground edges*, edges the body can move on, and *falling edges*, which the Graphbody falls down to get put on the next edge.



Graphbody B moves to the right on ground edge E_{G1}, falls down falling edge E_F and moves further on groundedge E_{G2}

Figure 1.2: Navigation Graph example

A Graphbody must be able to perform the following actions:

- move on a single ground edge
- transition to the next ground edge
- fall down falling edges
- collide with other bodies at a radius
- move on top of other bodies

Furthermore the simulation should follow these guidelines: Graphbodies should

- feel like they obey the laws of friction and gravity
- move smoothly and avoid instantaneous jumps in position

More ideas for extending the prototype are covered at various points of this paper, but these core building blocks are implemented in the prototype.

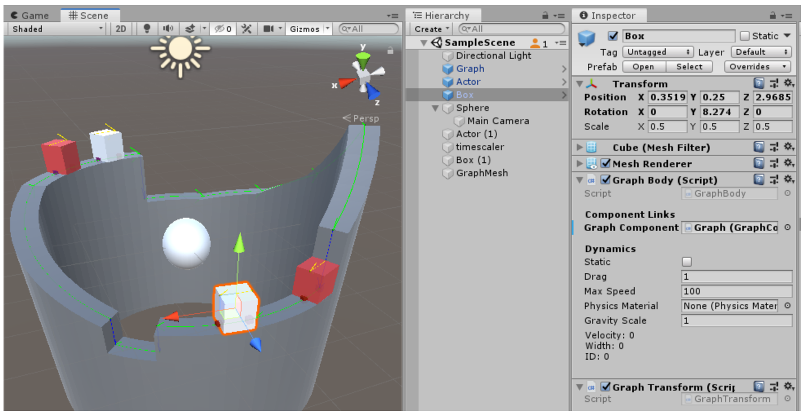
2 Theory

2.1 Game Engine Elements

A Scene or Level is the world all objects exist in. It can be interpreted as a three dimensional space that all Gameobjects are located in (like characters, boxes, the ground etc.) or as a hierarchy of Gameobjects. The hierarchy is known as a Scenegraph as introduced by Paul S. Strauss [SC92].

Gameobjects are all the objects in a game. They each have a parent, which is either the scene itself or another Gameobject. This also means that each GameObject has a variable number of Gameobjects as children.

Components are the functional building blocks that Gameobjects are made of. They are classes that each implement one behavior of the object e.g. visual representation, movement or storing some data.



```
SampleScene
  Directional Light
  Graph
  Actor
  Box
  Sphere
    Main Camera
  Actor (1)
  timescaler
  Box (1)
  GraphMesh
```

```
Box                                   Static
Tag  Untagged      Layer  Default
Prefab  Open  Select    Overrides
Transform
Position   X 0.3519  Y 0.25   Z 2.9685
Rotation   X 0       Y 8.274  Z 0
Scale      X 0.5     Y 0.5    Z 0.5
Cube (Mesh Filter)
Mesh Renderer
Graph Body (Script)
Script                GraphBody
Component Links
Graph Component       Graph (GraphCo
Dynamics
Static
Drag                  1
Max Speed             100
Physics Material      None (Physics Mater
Gravity Scale         1
Velocity: 0
Width: 0
ID: 0
Graph Transform (Scrip
Script                GraphTransform
```

left to right: 3D scene view, Gameobject hierarchy, Components of Box

Figure 2.1: Unity Scene

Transforms are special components. In games most Gameobjects use some implementation of a Transform that stores where the object is located in the scene. This Transform can be either in WorldSpace (the total position, rotation and scale in the world) or in LocalSpace (relative to some Object e.g. its parent).

DataType	Property	Description
Vector3	Position	The position of the objects center
Quaternion	Rotation	the objects orientation in space
Vector3	Scale	factor for the objects size

Table 2.1: Transform

2.2 Rigidbody Simulation

A Rigidbody is a solid body that cannot be deformed. James K. Hahn defined them as a set of physical properties (mass, coefficient of friction etc.) and a dynamic state (velocity, transform etc) [Hah88]. Using this separation the most relevant properties used in Unitys implementation are shown:

DataType	Property	Description
float	mass	resistance to change of linear motion
Vector3	angular mass	resistance to angular motion
Collider	Collider	shape
Transform	Transform	center of mass
Vector3	velocity	linear velocity
Vector3	angular velocity	angular velocity
float	friction coefficient	the amount of contact friction the body experiences
float	drag	factor for velocity each simulationstep
float	angular drag	factor for angular velocity each simulationstep

Table 2.2: RigidBody

The Rigidbody is updated in discrete timesteps, which are mostly uniform (the prototype uses the Unity default of 20ms). At each timestep the physics system calculates all forces acting on the bodies, changes their velocity (linear and angular) accordingly, and multiplies the drag on top of it. It then updates the bodies position and rotation, checks for collisions and reacts accordingly.

Colliders approximate the bodies visual shape as well as possible while maintaining performance when check for collisions (overlaps) with other colliders. The following table contains some of the most common colliders ordered from fast collision checks to best approximation of complex shapes.Axis aligned bounding boxes are excluded from the list, since they have some special properties if integrated into an object hierarchy.

2D Collider	3D Collider	Comment
Circle	Sphere	
2D Capsule	3D Capsule	pill shaped objects
Rectangle	Box	
convex Polygon	convex Mesh	meshes are arbitrary collections
Polygon	Mesh	of triangles surrounding a volume

Table 2.3: frequently used Collider

Checking these colliders for overlaps is generally a every computationally intense task and optimized using a hierarchy of colliders and collision areas. Having a more simple way to check collisions saves a lot of time for other computations.

Simulation Spaces As seen in table 2.3 Rigidbodies can be simulated in spaces of differing dimensions. Depending on whether the game features 2D or 3D gameplay, game developers typically chose to also simulate their bodies in that space.The graph-based approach assumes that most of the two dimensional simulations can also be approximated in a single dimensional space. A 1D space is a line. If all objects lack holes, they can be represented using line colliders. A line collider has a starting point and an end point between which is the space inside the collider. The benefits of a 1D system are due to the convenience and better performance of these line colliders. Another advantage is that any physical properties connected to rotation do not need to be simulated, since 1D objects cannot be rotated.

3 Third Party Tools

3.1 Unity

The unity game engine is a widespread tool for creating games. It is developed by Unity Technologies and offers tools for building game scenes, simulating 2D and 3D physics and rendering the result.

3D Physics in Unity are realized using Nividias PhysX [Cor], which is an open source realtime physics SDK. To simulate 3D Rigidbodies in Unity, a GameObject needs a Rigidbody Component and one of the Collider Components. The Rigidbody simulates the physical behavior of the object each time the "FixedUpdate()" loop is called, every 20 ms by default.

2D Physics in Unity are simulated using the Box2D SDK by Erin Catto [Cat]. It is one of the most widespread 2D physics engines in modern games. Similar to 3D Physics it is integrated using a Rigidbody2D component that uses one of the Collider2D Components for collisions. It also uses the same game loop for simulation.

4 Broad Design

4.1 Graph designs

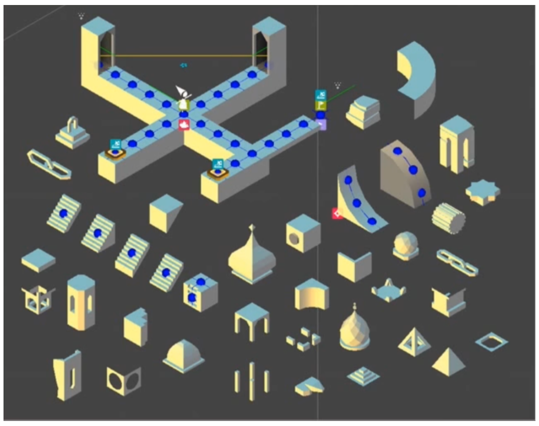
The design of representing bodies as objects on edges, as introduced in the target specifications, is not obvious excludes alternatives. The following paragraphs describe some of the possible alternative graph designs. Note that to fully understand all implications of the different designs, a full implementation is likely necessary.

4.1.1 Continuous and discrete Navigation Graph

The continuous Navigation Graph is the solution presented in this paper. It represents Graphbodies as objects on edges. Unlike the discrete Navigation Graph, the continuous design makes body-sizes, edge-lengths, step-sizes etc not limited to a set of predefined values. This offers the most amount of freedom, while still not supporting all possible scenarios. Bodies that stand on top of each other in particular may be better suited to a discrete variant. Even the continuous design requires limitations as shown in Section 6.2.2. Navigationgraphs are a perfect fit for 2D and 2.5D games. The prototype uses a design that has a clear definition for left and right movement, which is handy for character controllers in these games. It also allows for arbitrary placement of the vertices in 3D space. This makes transformed 2D designs like the cylinder in *Huge and Cute* possible.

4.1.2 Vertex Position Graphs

Graphbodies can also be represented as always staying on one vertex as shown in Figure 4.1a. This is a much more traditional use of graphs for movement and is used in many games such as *Lara Croft GO* and *Monument Valley*. It offers a lightweight solution for turn-based character controllers and can be made to appear continuous like in *Monument Valley* by placing the vertices close enough to each other and smoothing out transitions. It also offers a very fast solution for pathfinding. However this solution is not suited for smooth physical simulations. The concept of continuous velocities of bodies cannot be represented if bodies are stuck on the vertices. Nonetheless, further research in this field (e.g. creating a controller that allows these actors to stand on top of each other) could yield interesting new results.


(a) Vertex Position graphs in *Monument Valley*[Won]

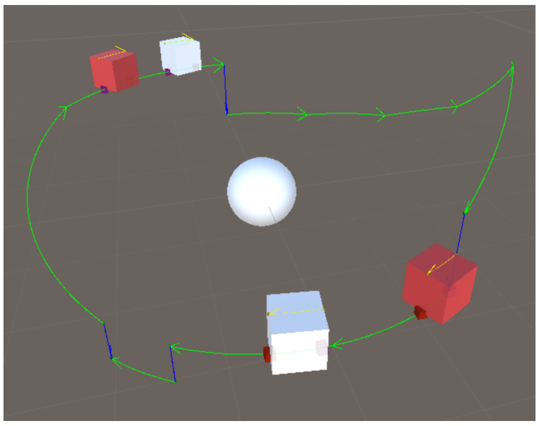

(b) Navigation Graph prototype with Edge Positions

Figure 4.1: Graph designs

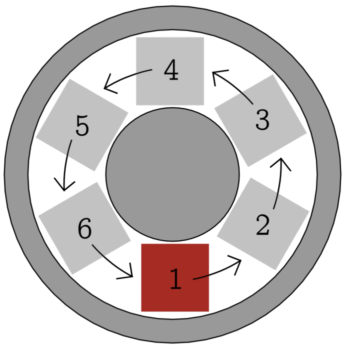
4.2 Collisions

This paper implements a collision approach that reverses the traditional way of handling them. Unitys Physics2D uses the order of operations introduced by James K. Hahn [Hah88] and solves the collisions after two Rigidbodies intersect instead of preventing intersections.

Post Intersection (e.g. Unity)	Pre Intersection (e.g. Graphphysics)
1. Simulate Movement 2. Find Collisions 3. Solve Collisions using Impulses	1. Calculate Distance 2. Move Other Body (if necessary) 3. Move Self

Table 4.1: Order of Operations for Collisions

The alternative is to check for collisions during the movement. Graphbodies can calculate the distance to the closest actor. If their desired movement exceeds that they recursively start a chain of movement requests that push all the Graphbodies along with them. This is only possible because the Navigationgraph is effectively a one-dimensional system. Therefore it can always compute a list of left and right neighbors. This list does not contain Graphbodies twice. Unlike Rigidbodies (see Figure 4.2) Graphbodies therefore rarely get stuck in loops, in which the Graphbody tries to push itself. The only exception are circular graphs that are fully filled with Graphbodies. In this case the Graphbody still stops due to a maximum push limit. More details are provided in Chapter 6.2.1. The reason why Graphbodies avoid most of these cases is that these movements can not be modeled with Graphbodies in the first place, since they are limited to left and right movement. This is the reason why collisions can be computed differently.



Side view: Rigidbody2D character moves (1) and pushes bodies (2-6) until pushing himself

Figure 4.2: Rigidbody movement loop

4.3 Graphbody designs

All of the following designs model the Graphbody as an object on an edge that has a certain width that is used for collisions with the terrain and other Graphbodies. They differ in the way they handle collisions with the terrain, visual representation, slope calculation and the way the distance between Graphbodies is calculated.

4.3.1 Definitions

Edgepositions are a pair of two attributes. The first one is the edge that the Graphbody is currently on. The second attribute stores the location on that edge. If it is in range [0,length(edge)] the Graphbody is still on the edge, otherwise it has an overhang. See Chapter 6.2.1 for a more detailed explanation.

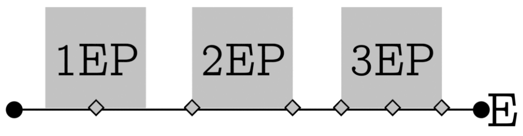


Figure 4.3: Graphbody designs

Single Edgeposition Graphbodies (1EP) store a single Edgeposition per Graphbody. This Edgeposition is located in the Graphbodies' center. The advantage of this design is its simplicity, since they only store the position and width of the Graphbody instead of storing redundant data.

Two Edgeposition Graphbodies (2EP) simulate two Edgepositions per Graphbody. This is the approach used in the prototype. They represent the outer border and can directly be used for checking collisions and calculating physical properties like slope. The 2EP Graphbody first moves the Edgeposition in front (e.g. for moving right move the right border first) to determine the distance it can travel. After that it can drag the other one behind by the same distance without a need to check for collisions.

Three Edgeposition Graphbodies (3EP) use three Edgepositions for the Simulation. Two are located at the left and right border of the object like in the previous design and one is located at the center of mass. This could be useful for better simulating events like falling or moving across convex vertices.

4.3.2 Comparison

To compare the performance of collided movement the 3EP design can be neglected. Its performance is equal to 2EP, because both simulate each border separately when moved. The following two metrics will be discussed: "How efficient is moving the Graphbody including collisions?" and "How difficult is it to find out where the Graphbodies borders are?". It is important to differenciate between *collided* and *uncollided* movements. Collided movement is required when the Graphbody needs to check if other Graphbodies are in its way. During uncollided movement the Graphbody can ignore those.

Moving a 2EP body requires to move the front border collided and then to move the other one without checking collisions (since the front border already moved all bodies that were blocking the path). After that both are available for Edgeposition lookups.

A 1EP body only needs to move its center collided. The problem with this design arises when doing positionlookups. To find its borders the Graphbody can either simulate one uncollided movement per lookup or cache the two positions. Caching requires 2 extra uncollided movements when moving the body. Both methods require more computations than the 2EP design and are therefore less efficient. The great advantage of this method is that it does not require collision margins for moving bodies. A 2EP design always needs to keep the borders at a certain distance to prevent the bodies from moving into each other, but since the width of the 3EP Graphbody is already preventing overlaps it does not need them.

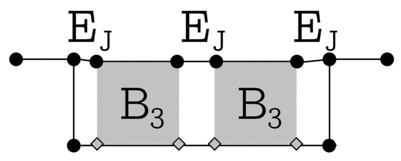
Bodytype	1EP	1EP cached	2EP/3EP
collided movement	M_C	$M_C + 2 \cdot M$	$M_C + M$
Edgeposition lookup	$M + L$	L	L

M = uncollided movement, M_C = collided movement, L = variable lookup

Table 4.2: Graphbody design performance estimation

4.4 Static and Dynamic Graph

One of the goals of the graph-based approach is to enable Graphbodies to stand on top of each other. To achieve this two types of edges and vertices are introduced here.



Dynamic Graph including Vertices on top corners of Graphbodies and jump edges E_J connecting them

Figure 4.4: Dynamic Graph

Static Vertices and Edges have a position or length, that is not dependent on other Objects. While it would be possible to move them at runtime, this would require a manual update of all the adjacent objects and is not explained here.

Dynamic Vertices and Edges have propeties that depend on other objects and change during runtime. The basic idea is that each Graphbody has two dynamic vertices. One on its top-left corner and one on the top-right. Both are connected by a dynamic edge that other Graphbodies can be on as shown in Figure 4.4. Note that this implies that the position of the vertices is dependent on the position of the Graphbody.

To get on top of another body a Graphbody uses so called *jump edges*. These connect the tops of all Graphbodies and ledges (startpositions of fall edges). So if a Graphbody is going to fall, it first checks for any nearby Graphbodies that it might need to jump on.

5 Architecture

5.1 Layers

The prototype uses Unity's component system. This chapter will show how these components are organized in layers and how these layers are interacting with each other. The architecture is ordered in the following layers:

Layer	Function	Components
Visual Layer	display positions, smooth movement	Graphtransform
Physics Layer	simulate gravity, friction etc	Graphbody
Graph Layer	handle movement on graph	GraphComponent
Data Layer	store data, fast data access	GraphContainer, GraphTransform
No Layer	Graph generation and visualization	GraphCreator, GraphVisualizer

Table 5.1: Architecture layers

The GraphTransform appears in two layers, since it is on the ne hand storing the Graphbodies position in Graphspace (see Section 6.1.1), but also handles the smooth movement of the visual representation. The difficult connection between the GraphContainer and GraphTransform is one of the flaws of the design presented here.

5.2 Information Flow

To demonstrate how the typical flow of information travels through these layers, the following paragraph describes the typical scenario of simulating a Graphbody for one tick. Note that most processes here are simplified and will be explained in more detail in Chapter 6.

The Graphbody starts the simulation. Its Simulate() method is called by Unitys FixedUpdate() loop, which is typically triggered every 20ms. The Graphbody now takes into account its last velocity, gravity and friction to determine its current velocity. Based on the duration of the timestep and its velocity it calculates the desired distance it should move to the left or right. To actually move the borders it sends its desired distance and its current position, which is stored in its own GraphTransform, to the layer below it.

The Graphcomponent is now challenged with determining if the desired movement distance is possible and where the Graphbody will end up. It first tries to move the position along the current edge. The only object that can block the movement on a single edge are other Grahbodies. So the first data the Graphcomponent requests is the position of all Graphbodies on his edge. If it is able to reach the vertex of his current edge, it needs to decide which edge is the next one it transitions to. This is realized through a series of requests to the lowest Layer.

The Graphcontainer tries to resolve these requests. These are realized by looking up a key (Vertex, EdgeType, Movement direction) in a dictionary. These dictionaries are generated when starting the application and provide a fast method for finding the next edge. Afterwards the information moves up the entire system. The Graphcomponent solves the FindNextEdge Request, the Graphcomponent returns the new position and the GraphBody updates its GraphTransform accordingly. The Graphbody then also requests a visual update from the Graphtransform.

The Graphtransform directly accesses the GraphContainer to find the positions of the vertices of the edge it is currently on. It interpolates between these points using the Edgepositions to determine the target location and rotation. This target is smoothed for better visualization of instant movements and set in the gameobjects transform.

6 Implementation

6.1 Data Layer

6.1.1 Definitions

GraphSpace is the Space defined by the Navigation Graph. Positions in Graphspace are always on an edge. Transitions between edges are determined by the vertices and the edges connected to them. The Graphspace has the two directions left and right. Movement in one of the two directions from one position in Graphspace over a certain distance results in a new position in Graphspace.

EdgePositions are structs that represent a position in Graphspace. They store an EdgeID and a position on that edge. The EdgeID consists of a boolean determining whether the edge is stored in the dynamic or the static edge list and the index in that list.

6.1.2 Graphtransform

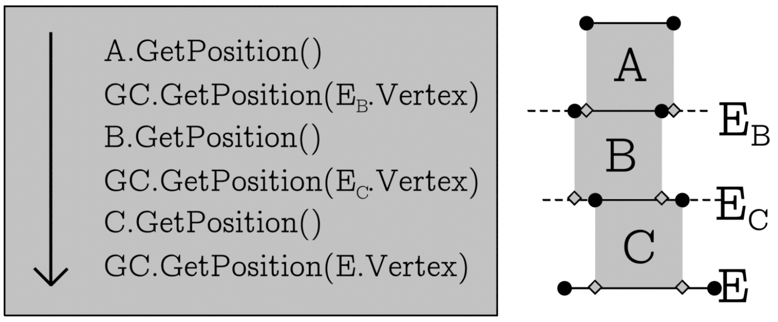
The Graphtransform stores the following data:

GraphTransform		Purpose
- leftEdgePos:	EdgePosition	Left Border Position in GraphSpace
- rightEdgePos:	EdgePosition	Right Border Position in GraphSpace
- height:	float	vertical visual offset
- graphComponent:	GraphComponent	link to Graph
- fallTime:	float	last start of Fall for Visual Layer

Table 6.1: GraphTransform

Its purpose is to provide easy access to its position (in 3D space and Graphspace) and to update the visual representation accordingly. While determining its position in Graphspace is just a matter of reading out the EdgePositions, finding its Position in 3D space is more difficult. For example, a Graphbody A stands on top of the Graphbody B. B also stands on a Graphbody C, which stands on a static edge E. Finding the

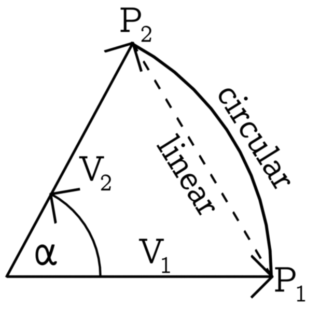
3D position of *A* requires knowledge of the 3D position of *B* which requires the 3D position of *C* (see Figure 6.1). This demonstrates how chains of position lookups can occur. These get resolved through a recursive call, which eventually leads to a static edge.



```
A.GetPosition()
GC.GetPosition(E_B.Vertex)
B.GetPosition()
GC.GetPosition(E_C.Vertex)
C.GetPosition()
GC.GetPosition(E.Vertex)
```

Figure 6.1: Position Lookup Example

This example already emphasizes that many position lookups are run multiple times without moving the body, since other Graphbodies need to know their 3D position as well. This means that the current position in 3D space could be cached until the body is moved again. While this is currently not implemented in the prototype, it is an easy addition to improve efficiency.



Interpolation between Points P_1 and P_2 by rotating Vector V_1 towards V_2

Figure 6.2: Circular Interpolation

Another feature of the GraphTransform is its ability to implement different interpolation-methods for vertex positions (see Figure 6.2). The standard interpolation is linear, but for circular levels the implementation also supports a circular interpolation. This special interpolation works by interpreting the start and end position as two vectors starting at (0,0,0). Rotating from one to the other, while interpolating the length linearly, returns the interpolated position. Furthermore, there is potential in creating more interpolation methods (e.g. using splines) for different types of games.

6.1.3 Graphcontainer

Data

The Graphcontainer stores the following data:

GraphContainer		Purpose
- Vertices:	List<Vertex>	Static Vertices
- Edges:	List<Edge>	Static Edges
- DynamicVertices:	List<Vertex>	Dynamic Vertices
- DynamicEdges:	List<Edge>	Dynamic Edges
- edgeIDDict:	Dictionary<Edgekey,List<ID> >	Fast Edge Lookup
+ IsCirlce:	bool	Circle Interpolation
+ GraphBodies:	List<GraphBody>	Graphbodies for Collisions

Table 6.2: GraphContainer

The dynamic and static vertices and edges are stored in separate lists. As mentioned in Section 4.4 this is due to the way their properties are accessed. For example the GraphContainer provides a method to access vertex positions. It takes a VertexID as input, which like the EdgeID is composed of a boolean that defines whether the vertex is static or dynamic and the index in the corresponding list. If it is static the GraphContainer can directly access the value and return it, while for dynamic vertices it looks up their position through the chain described in the previous chapter.

Edge Lookup

An important task of the Graphcontainer is finding edges that are attached to specific vertices. If a Graphbody was moving to the left and landed on the end of an edge, it would then be looking for another ground edge to the left. These informations (Vertex, Direction, EdgeType) are used to compose a key, so the GraphContainer can look it up in a dictionary in almost $O(1)$ [Mic]. This dictionary is created on startup to provide very fast access. Note that this is one of the larger design flaws currently, since this means it is difficult to add and remove Graphbodies during runtime. A direct implementation of this dictionary prevents it from adding multiple edges of the same type to one key. To make this possible the dictionary links to a list of edges instead of single edge. This is necessary to enable each Graphbody to have a jump edge to each ledge. The decision which of the edges is used is illustrated in Section 6.2.1.

Graph Creation

In the following sections all static edges and vertices will be refered to as the static graph as opposed to the dynamic graph, which contains all dynamic vertices and edges. Creating the entire Navigation Graph takes the following steps:

1. Editor Time
   a) create static Graph
   b) link GraphBodies
2. On Start
   a) add static Graph to Dictionary
   b) build dynamic Graph
   c) add dynamic Graph to Dictionary

The Static Graph is generated in the Editor by classes that inherit from GraphCreator. The prototype only uses a SimpleGraphCreator that takes in a list of vertices and connects those. It decides automatically whether an edge is a ground or falling edge depending on its steepness. While this is a very easy way to generate graphs, it does not cover all possible cases. For example levels with floating platforms can not be generated using this approach. A Navigation Graph for these levels could potentially be created using the already existing geometry or colliders in the scene, similarly to how it is already being done with Navmeshes in Unity. While this approach requires a new large system it would help level designers, who work with the Navigation Graphs.

The Dynamic Graph is generated later at Startup. It connects ledges (Vertices that are at the top of a falling edge) with the dynamic vertices on top of Graphbodies in three different ways:

1. Ledge - Graphbody Top
2. Graphbody Top - same Graphbody Top
3. Graphbody Top - other Graphbody Top

This way Graphbodies can jump from ledges on other Graphbodies, stay there and jump to the next Graphbody. The dynamic vertices for the top of the Graphbodies are currently indexed in a way that makes it easy to find out which Grahbody they belong to. Since all Graphbodies already have an index N in their list, their corresponding dynamic vertices are indexed with $(2 \cdot N)$ and $(2 \cdot N + 1)$. This link should be made more flexible in the future, e.g. by storing a Graphbody link in each vertex, to make it easier to add and delete Graphbodies at runtime.

6.2 Graph Layer

6.2.1 Movement in Graphspace

The GraphComponent always uses a target distance that it tries to to move the Edge-Position for. Since some cases (e.g. running into a wall) require to stop the movement before the target is reached the movement method always returns the new EdgePosition and the actual distance moved for further processing. The following cases show how the Graphbody is supposed to move or stop in different cases.

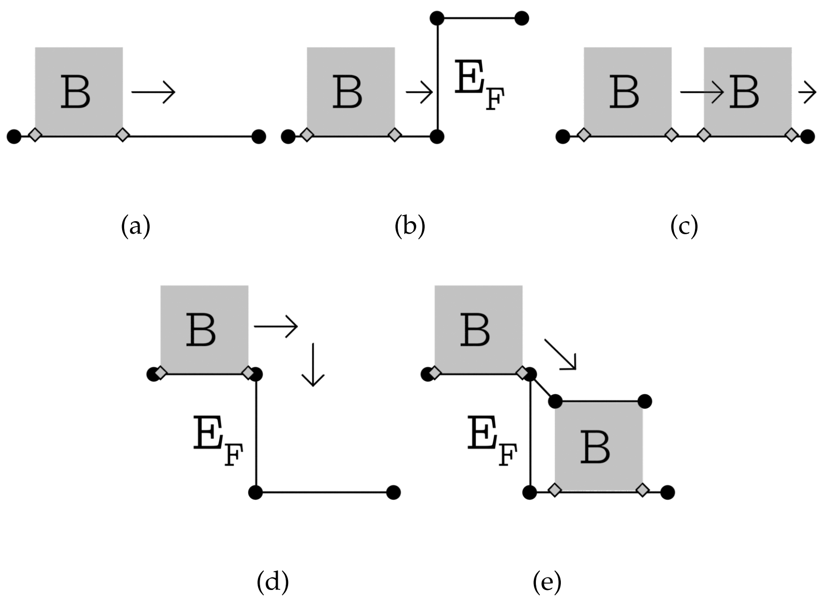


Figure 6.3: Movement Cases

Expected Behaviour

(a) On an unobstructed Path the EdgePosition simply increases or decreases its position on the current ground edge. If it exceeds the current edge and a next ground edge is adjacent to the current one, it can simply move onto that for the remaining distance.

(b) Running into a wall is the same case as not finding a next ground edge to go on. In this case the movement is cancelled at the end of the edge and the actual distance is returned.

(c) On collision with Graphbodies the EgdePosition should move all the Graphbodies in its way as far as possible without slowing down. If these Graphbodies have to stop (e.g. because they run into a wall) it also stops before reaching the target distance.

(d) When reaching an unobstructed ledge the EdgePosition is first moved into an overhang. This means it still stays on the edge, but exceeds the [0,EdgeLength] range. When both EdgePositions of the Graphbody are in an overhang, the Rigidbody falls and both EdgePositions are moved to the ground edge after the fall edge.

(e) When reaching an obstructed ledge the Graphbody still goes into overhang and then instead of using the fall edge it uses the jump edge from that ledge to the Graphbody that is obstructing the fall.

To split these cases into more managable chunks the GraphComponent implements two methods. GetNextEdgeFallPosition() handles jumps and falls, while GetNextEdgePosition() is responsible for the rest. So the Graphbody first moves. If it afterwards is in an overhang, it decides which jump/fall edge to use and uses that.

Movement on Ground Edges

The GetNextEdgePosition() method tries to move the starting EdgePosition along ground edges until it either hits an obstacle or goes into overhang for a jump/fall. While doing this it takes the decisions and actions in Figure 6.4. The first half of the tree ensures it moves properly on the first edge, the second half handles movement on the next ground edge or movement into an overhang. Each collided movement within one edge is handled by calling MoveOnEdgeCollided(). That method checks for other Graphbodies on the current edge and the next one and ensures the Graphbody does not move closer to the other body than the collision margin. This is necessary to ensure that vertices do not end up in the same place.

The prototype searches for other Graphbodies by simply going through the list of all Graphbodies and checking their position. Note that this is inefficient and could be solved by using one of two methods. The first approach is the same used by Rigidbodies. The idea is to split up the graph into chunks that each contain some edges and then only checking Graphbodies that are in these (or adjacent) chunks. The other approach is to maintain a datastructure that saves the left and right neighbor for each Graphbody. These can change, so it needs to be updated regularly, but this way not every Graphbody has to find its neighbors, instead it can just be looked it up in the global structure.

Furthermore disconnecting the collision check from the movement could help, since it does not need to be called twice when transitioning from one edge to the next. The collision check already includes the next ground edge to avoid moving into other bodies that are right at the start of the next edge. Removing this redundancy increases performance.

Falling and Jumping

If the Graphbody is about to fall it needs to decide which jump edge or fall edge to use. So it evaluates all jump edges that are connected to the ledge and checks if their horizontal distance is smaller than the Graphbodies width. This is a good proxy to determining whether the ground edge at the end of the fall edge is obstructed. But it is recommended to check that directly, since it handles certain cases with tilted ground edges more accurately. If no jump edge with this property exists, the ground edge is unobstructed and the Graphbody can fall down. If any jump edges were found, all of them that move the Graphbody upward are ignored. If no jump edge is left after that step, the body is blocked and cant move on. If one or multiple jump edges are left the one with the smallest downward distance is picked.

6.2.2 supported Cases and outlook

The rules described here are only the foundation needed to create any thinkable 2D level. Graphbodies really are stuck in the one dimensional space and therefore can not easily check for collisions with objects that are not directly on their string of edges. The following sections will show some of these scenarios, help with a few guidelines that

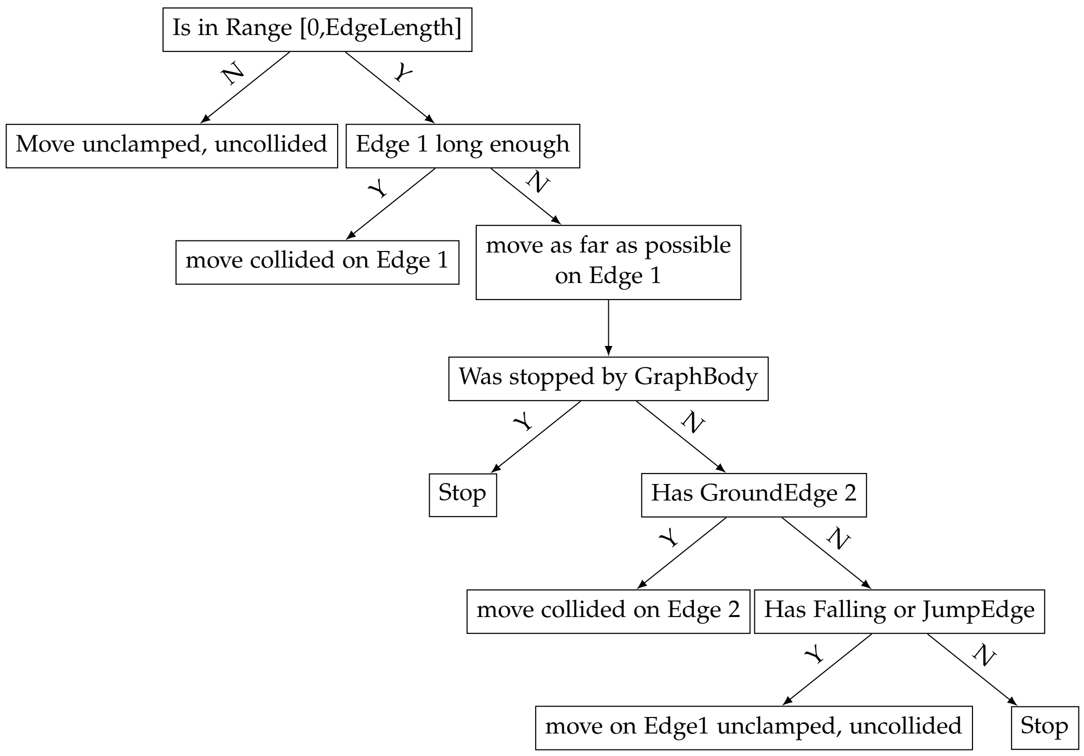


Figure 6.4: Decisions in GraphComponent.GetNextEdgePosition()

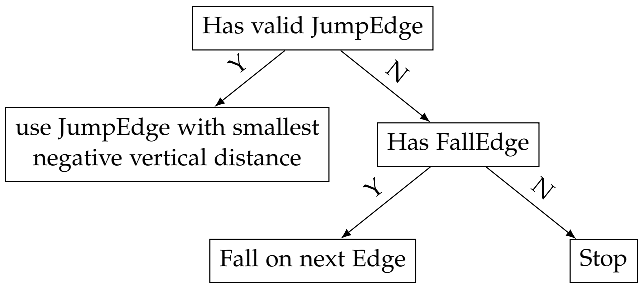


Figure 6.5: Decisions in GraphComponent.GetNextEdgeFallingPosition()

reduce or even prevent unexpected behavior and show how some of the problematic scenarios could be resolved.

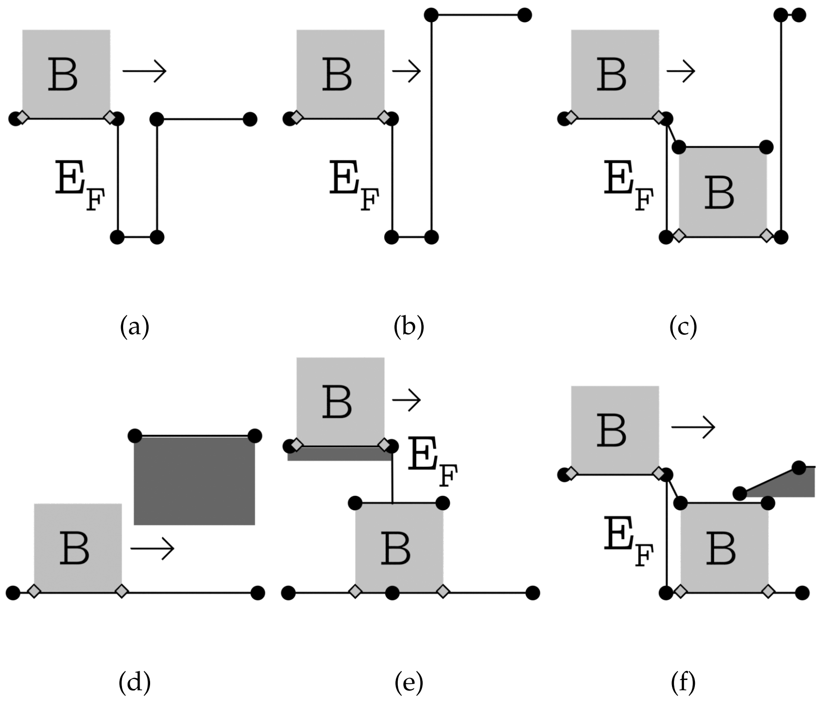


Figure 6.6: Unsupported Case Examples

Problematic Cases

(a) Short pits are still a problem using the current rules, because ledges do not get connected with jump edges to each other. This could be resolved by changing the dynamic graph generation to include these additional jump edges.

(b,c) Short pit with height differences are more problematic. The Graphbody assumes that it can extend to its full overhang and therefore starts to intersect with the right wall. This could be fixed by adding a maximum overhang to each ledge. As shown in (c) this can also happen when staying on other Graphbodies. Solving this is a much more serious problem that might involve knowing the width of each pit and calculating the leftover overhang.

(d) Tunnels are not supported with the current rules. This could be fixed by adding a height parameter to each edge and preventing Graphbodies that are higher than that to move onto them.

(e) Falling on the center of Graphbodies is also not supported. If edges as shown in this example were never blocked it would be easy to use the jump edge to fall into the middle of the ground, but when the area is blocked a different solution is required.

(f) Sharp Ledges pose a challenge, since Graphbodies can not jump onto them while they are in the middle of the top of another Graphbody. This still requires a new solution.

Guidelines

While the following list cannot guarantee that there are no complications it removes a lot of the unwanted cases including some of the ones shown above. This requires further exploration that does not fall within the scope of this paper. However, what follows is a preliminary set of rules:

- Keep a uniform body size
- Keep fall edges perfectly vertical and make then one body height tall
- Let fall edges only end in perfectly horizontal ground edges
- Make all ground edges at least one body length long
- Only have a single layer of static edges

6.3 Physics Layer

6.3.1 Graphbody

The Physics layer controls the distance each Graphbody is supposed to move each timestep. Graphbodies can be either static, meaning their velocity is controlled by setting it by some custom logic, or dynamic, which automatically simulates forces like gravity and friction. This velocity is clamped using a lower and upper bound. If the velocity is lower than the minimum velocity it gets set to 0, to avoid unnecessary simulations. Similarly to Unitys Rigidbody2D the GraphBody has a drag value that gets multiplied on the velocity in each timestep.

6.3.2 Friction and Gravity

Friction and Gravity are the two forces that act on all dynamic Graphbodies. In the prototype the bodies mass is assumed to be 1kg (defining unities standard length unit as 1m). Figure 6.7 shows the prototypes implementation of calculating the forces that act on the GraphBody due to friction and gravity based on the design shown in the book Game Engine Architecture [Gre17].

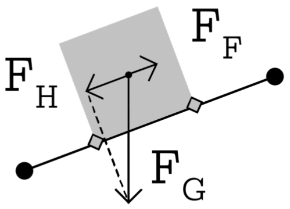
```
//simulate gravity
float gravity = _GravityConstant * _GravityScale;
float steepness = GraphTransform.getSteepness();
velocity += steepness * gravity * Time.fixedDeltaTime;
//simulate friction
float frictionCoefficient = 0;
if (m_PhysicsMaterial)
{
frictionCoefficient = m_PhysicsMaterial.friction;
}
float friction = (1.0f - Mathf.Abs(steepness)) * frictionCoefficient * gravity;
velocity = Mathf.Max(Mathf.Abs(velocity) - (friction * Time.fixedDeltaTime), 0)
        * Mathf.Sign(velocity);
```

Figure 6.7: Friction and Gravity in GraphBody.FixedUpdate()

GraphTransform.getSteepness() returns the dot product between the up vector and the vector that points from one EdgePosition in 3D space to the other and gets signed to represent the movement to the left or right direction. Multiplying this with the gravitational force yields the amount of force along the grounds tangent. After that the dynamic friction is subtracted, which is dependent on the amount of force that presses the body onto the surface. This can be calculated that using $(1 - steepness) \cdot gravity$ since it is the gravitational force along the surfaces normal. Note that this does not simulate static friction, which would be much higher.

6.3.3 Evaluation

To confirm the prototype it was directly compared to Unitys RigidBody2D. The tests use an inclined edge, represented by a polygon collider for the Rigidbody and a Graphbody with a corresponding Rigidbody (that uses a Boxcollider to mimic the Graphbodies shape). The bodies start at the bottom of the edge, get shot upwards by setting their velocity and slide down. The tests used the following parameters:

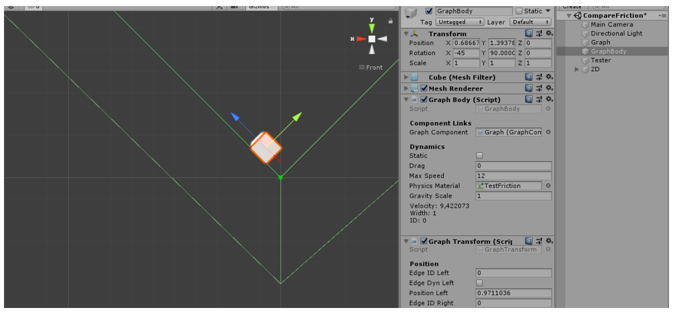


Gravity F_G gets reduced along the ground-normal to F_H, Friction F_F

Figure 6.8: Gravity and friction at slope

Parameter	Description
inclination	inclination of the edge in degrees
friction	friction coefficient of both bodies
speed	start velocity along the edges normal of both Bodies
gravity	gravity constant of the GraphBody

Table 6.3: Physics Test Parameters

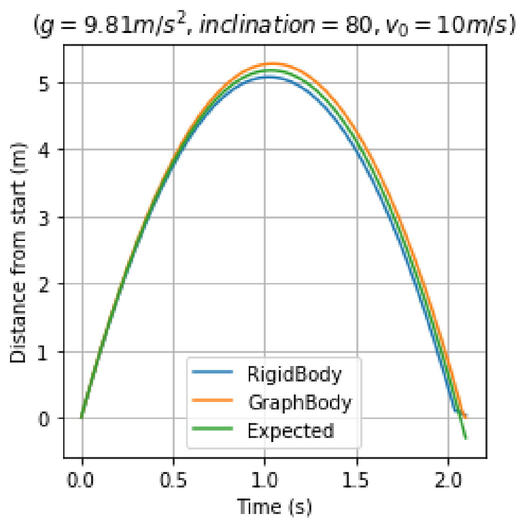

```
GraphBody                  Static
Tag Untagged   Layer Default
Transform
Position  X 0.68667  Y 1.39378  Z 0
Rotation  X -45      Y 90.0000  Z 0
Scale     X 1        Y 1        Z 1
Cube (Mesh Filter)
Mesh Renderer
Graph Body (Script)
Script            GraphBody
Component Links
Graph Component   Graph (GraphCom
Dynamics
Static
Drag              0
Max Speed         12
Physics Material  TestFriction
Gravity Scale     1
Velocity: 9,422073
Width: 1
ID: 0
Graph Transform (Scri
Script            GraphTransform
Position
Edge ID Left      0
Edge Dyn Left
Position Left     0.9711036
Edge ID Right     0
```

The Graphbody (orange outline) and Rigidbody get pushed along the ground (green) with an initial force (red)

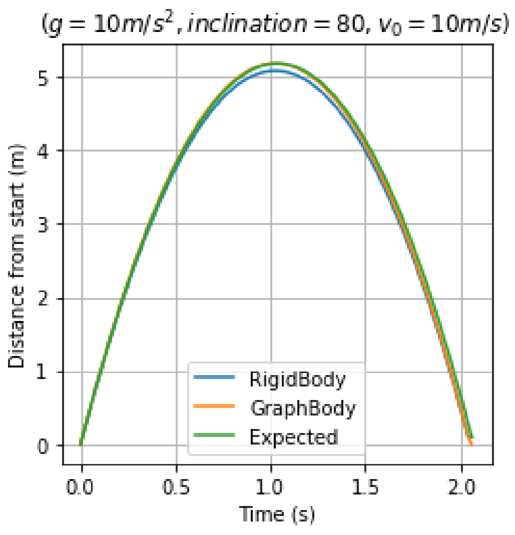
Figure 6.9: Gravity Test in Unity

Gravity Test

The Gravity test uses a friction of zero to eliminate the influence friction has on the movement. The prototype was tested at different inclinations and speeds. It turned out that the prototype did not perform optimally with gravity set to 9.81, which is the value used by Unities 2D physics by default and in the tests. The best results were archieved with a gravity value of 10 (as seen in Figure 6.10). This might be due to the different collision solving approaches, since both systems work very similar in all other regards. But since the differences are neglectably small, the gravity implementation was successful.



(a) $g = 9.81$

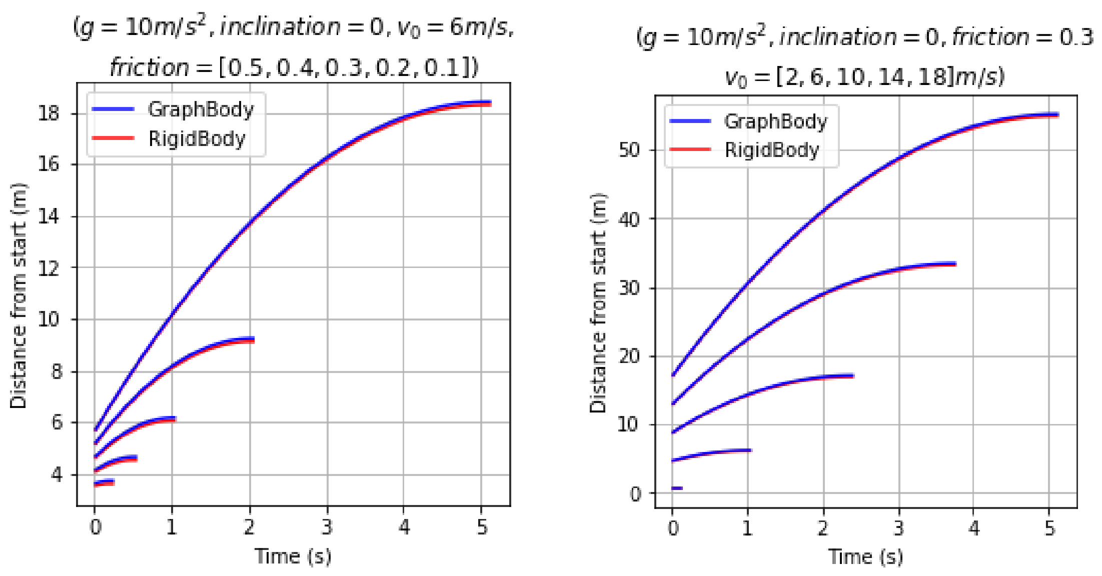


(b) $g = 10$

Figure 6.10: Gravity Test

Friction Test

The friction test used a flat plane to simply slide the Bodies along until they stop. These tests had to be run one at a time, since Unity seems to have trouble to update the friction coefficient of graphbodies during runtime. The plots in Figure 6.11 show two test series. The tests from 6.11a have different friction coefficients, while 6.11b uses different starting velocities. As demonstrated both plots show a high similarity to Unitys implementation. This was an unexpected result, since Box2D is supposed to use static friction, which is missing from the prototype. Even though the Box2D documentation states that they are both (static and dynamic friction) implemented using the same coefficient, no measurable difference was shown in the test. The difference appears to be neglecible for slow moving bodies and likely only applys to non moving bodies.



(a) variable friction

(b) variable velocity

Figure 6.11: Friction Test

6.3.4 Physics layer outlook

Variable Mass is as mentioned still missing. This can be fixed by adding a mass value to each Graphbody and taking it into account when simulating gravity and friction. This also opens up the possibility to add a AddForce() method similar to the one implemented in Unitys Rigidbodies.

Collisions options are another missing feature. In the prototype Graphbody-GraphBody collisions double the impulse by pushing the other body in front of it at full speed. This works well for puzzle games like *Huge and Cute*, but does not support a wide range of applications. One approach would be to detect collisions in the GraphComponent during movement, return them to the Graphbody and let it simulate the results. This would keep the domain of responsibility for each layer.

EdgeFriction is a feature that would help to simulate different ground types such as ice (low friction) or grass (high friction). The prototype only uses the GraphBodies friction coefficient to decide how much it needs to slow down. In classic Rigidbody simulations two coefficients are mixed to determine the friction between two objects. The same idea can be applied here, by storing a friction coefficient for each edge.

6.4 Visual Layer

6.4.1 Smooth Falling

The visual layer is there to smooth out instant movements that happen at simulation level. In the prototype it only handles jumps. Jumps are instant because it simplifies collision checks drastically. The downside is that bodies look as if they are teleporting from the beginning of the jumpedge to its end. To solve this issue the visual representation of the body is slowly moved down, while the EdgePositions are already moved to the next ground edge as shown in Figure 6.12.

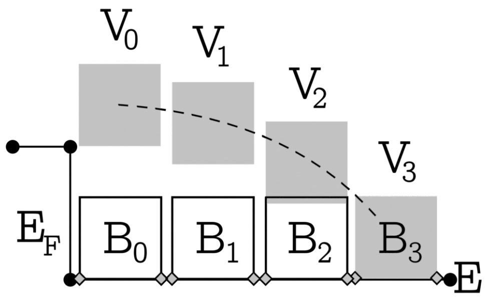


Figure 6.12: Falling GraphBody

Instead of setting the visual position instantly, only its horizontal component (in localspace) is set to be aligned with the new position. Furthermore the vertical axis stays unchanged and only moves towards the target by $(fallTime \cdot fallSpeed)$, where fallTime is the time since the fall has started and fallSpeed is a global value to tweak the visual gravity. This way the visual representation accelerates towards the target and is then stopped by using the target as a lower bound.

6.4.2 Visual layer outlook

Smooth Rotations are missing from the implementation. The same ideas that were used for the position can also be applied here. The rotation should adjust with a constant speed towards its target if a fall was triggered recently.

Smooth Jumps are only necessary in cases where the edges are not perpendicular to another or the height of the jump is not equal to zero. In that case it behaves similar to a Jump and should be triggered.

Animations may not be necessary for cubes, but when adding characters this addition could be included in the visual layer. Especially since it is already tracking jumps and falls it would be a perfect place to integrate an animation system or to expose more information to enable other scripts to access it for that use.

7 Evaluation

7.1 Performance

Before evaluating the performance it must be noted that the working prototype is neither comprehensive, nor completely optimized. The improvements suggested in Chapter 6 will increase the performance drastically and help make it more competitive. This is why a direct comparison between Unitys system and the Navigation Graph is not included in this paper.

7.1.1 Collisions

One of the important metrics is the performance of the collision system. The following test uses a set of horizontalground edges that are combined in one large ground. On that ground one character GraphBody is placed that is moving to the left and n boxes that the character will pick up one after another and push in front of it. This is close to a worst case, since it leads to long movement chains. The character pushes Box 1, which pushes Box 2 and so on. The time used for the movement was measured using CSharps Stopwatch class in System.Diagnostics. Note that stopping the time brings its own constant overhead, but since time growth is the more important measure, it can easily be ignored.

The computation effort grows linearly over time as seen in Figure 7.1. This conforms to expectations, since each box needs to run a similiar movement. However, the data also shows consistent spikes, despite the test being run 100 times to reduce the influence of any other programs that use the CPU. These could be due to other complications on operating system level or certain events, like many boxes getting pushed over to the next edge at the same time. But the data clearly shows a linear relation between the number of boxes being pushed and the time needed to compute the movement.

Figure 7.2 shows a more critical issue. It shows the amount of computation time needed for exactly the same action, but with a different amount of boxes in the scene. The graph shows the time it needed to do a single movement step after one second in the box push test with a variable amount of boxes in the scene. The data shows

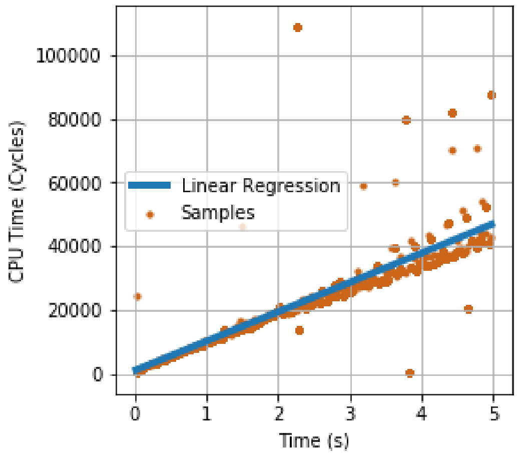


Figure 7.1: Performance over Time in pushing boxes test

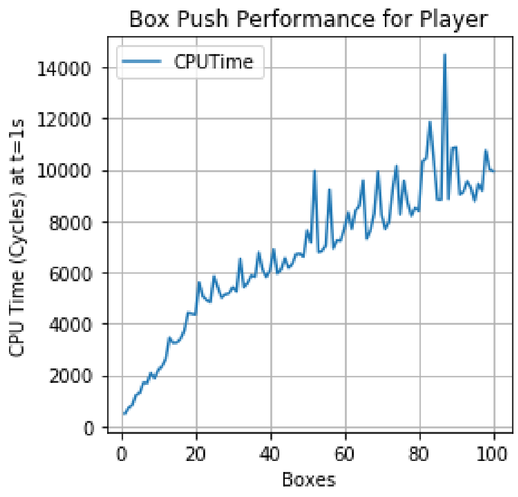


Figure 7.2: Performance over number of boxes in pushing boxes test

how it rapidly grows until the number is equal to 20, which is the amount of boxes the character is pushing after one second. But it still grows linearly after that. Due to the fact that all boxes get checked for collision, the more boxes there are, the more difficult it is to push a fixed amount. It can be solved as is outlined in Section 2.2, but it is a necessary measure to be improved before the system gets taken into regular use.

7.2 Realism and Predictability

As shown in Section 6.3.3 the movement follows traditional Rigidbody physics very closely and therefore feels realistic. Even the instant movement when falling is smoothed out. Only smoothing out the other instant movement cases (like jumps) and other physics features can increase realism further.

Predictability is the metric that measures the difference between player expectations and the actual behavior of the simulation. A comprehensive user survey is out of the scope of this paper, but three issues can be singled out.

Pushing multiple boxes into a wall. In this, the prototype yields much better results than the traditional Rigidbody method. Due to only being aware of their direct neighbors, Rigidbodies are incapable of solving this issue, while the presented approach can evaluate all Graphbodies in one movement. Rigidbodies therefore tend to lead to shaky and unpredictable behavior.

Movement at constant speed. RigidBodies have difficulty with this task even if the velocity is set at every timestep, because they always experience some deceleration by pushing other boxes. The Navigation Graph prototype is designed for cases where the character needs to not get slowed down in order to create more predictable behavior.

Unsupported cases. Since e.g. falling off of Graphbodies is not supported yet, the player might expect that behavior, but then can not do it. The same problem arises for all the other cases mentioned in Section 6.2.2. The great advantage of Rigidbodies is that they always behave in the same predictable way.

7.3 Software Architecture

It is vital to give prospective users of a graph-based approach an easy way of understanding, using and potentially expanding the system. The prototype in its current state does not lend itself to be used without complication. However, there are multiple options to improve this project.

Most notably, the classes tend to contain long and incomprehensible methods. While that is an acceptable disadvantage in a prototype such as the one created for this paper, it must be remedied to be taken into regular use. To achieve this it is recommended to split up existing classes and put more emphasis on modularity and encapsulation.

A further adjustment to be made is a move of Graphbodies into the Graphcomponent, in order to turn the Graphcontainer into a mere data storage for the Graph itself, so that it can be reused in multiple scenes. Another possibility is a split of the dynamic and the static graph into two separate containers that are managed by a third class. That may allow the scene to hold the GraphContainerManager, the Dynamic Graph created during startup and only link to the static Graphcontainer.

Further the Graphcomponent could be split into two classes. The first one should be a Component for managing links to all the other objects (Graphbodies, GraphContainer, etc.) and the other one a class that does all the calculations of the Graph Layer.

8 Conclusion

The Navigation Graph design is a first step towards understanding all possibilities graph based character controllers have to offer. In particular the prototype revealed the flaws of the approach. As stated in the introduction the following features were part of our requirements:

- let Graphbodies fall down fall edges
- let Graphbodies collide
- let Graphbodies have a non 0 width

The sum of all three requirements led to a fourth one. Stacking Graphbodies on top of each other to deal with the complications blocked ledges brought. The set of these four features created the many difficult cases shown in Section 6.2.2.

8.0.1 Alternative designs

To encourage further research around this topic the following list shows some alternative design concepts.

No Fall edges. After removing fall edges from the current design, the resulting graph is a network of ground edges. Simulating Graphbodies on these including collisions and physics is much easier and does not lead to large complications. One of the interesting challenges might be to optimize the collision system and maybe to build a system that allows for forking the ground edges.

No collisions. This is a great design for point and click adventure games. Generally the characters in these games can just walk through each other. Without the need for collisions, body stacking is not necessary.

No width. This is similar to the previous design. Characters can visually overlap as long as they don't change their order. This is already used in games like *Old Man's Journey*, which includes puzzles with sheep that need to be moved out of the way of the main character, because he cannot walk through them.

Traditional Collisions. One unexplored approach might be a hybrid system of traditional collisions and the graph-based movement. When the Graphbodies have full awareness of their 2 dimensional environments they can react to many of the cases that are problematic in the current prototype.

Vertex Graph Body stacking. As mentioned before a hybrid of using the vertex graphs from *Monument Valley* and adding body stacking is a promising idea. While these systems cannot simulate physics, stacking Graphbodies in them and correctly handling all connections is a challenge on its own.

8.1 Choosing the right system

When considering Navigation Graphs for a game, it is recommended to only use the approach shown in this paper if the levels have a simple structure. If possible one of the three requirements shown above should be avoided or Rigidbodies might deliver better results. The implementation of the Navigation Graph prototype showed that there is a multitude of different designs to chose from and that further research is necessary to show the complete picture.

List of Figures

List of Tables

Bibliography

[Cat] E. Catto. *Box2D Documentation*. URL: `https://box2d.org/documentation/index.html` (visited on 03/04/2020).

[Cor] N. Corporation. *PhysX Documentation*. URL: `https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Index.html` (visited on 03/04/2020).

[Gre17] J. Gregory. *Game Engine Architecture, 2nd Edition*. A K Peters/CRC Press, 2017. Chap. 12.4.7.4.

[Hah88] J. K. Hahn. "Realistic animation of rigid bodies." In: *Acm Siggraph Computer Graphics* 22.4 (1988), pp. 299–308.

[Mic] Microsoft. *.NET Framework 4.8 Documentation*. URL: `https://docs.microsoft.com/de-de/dotnet/api/system.collections.generic.dictionary-2?view=netframework-4.8` (visited on 03/09/2020).

[SC92] P. S. Strauss and R. Carey. "An object-oriented 3D graphics toolkit." In: *ACM SIGGRAPH Computer Graphics* 26.2 (1992), pp. 341–349.

[Won] K. Wong. *The Art of Monument Valley*. URL: `https://youtu.be/i0X8-5PpYVg?t=1339` (visited on 03/10/2020).