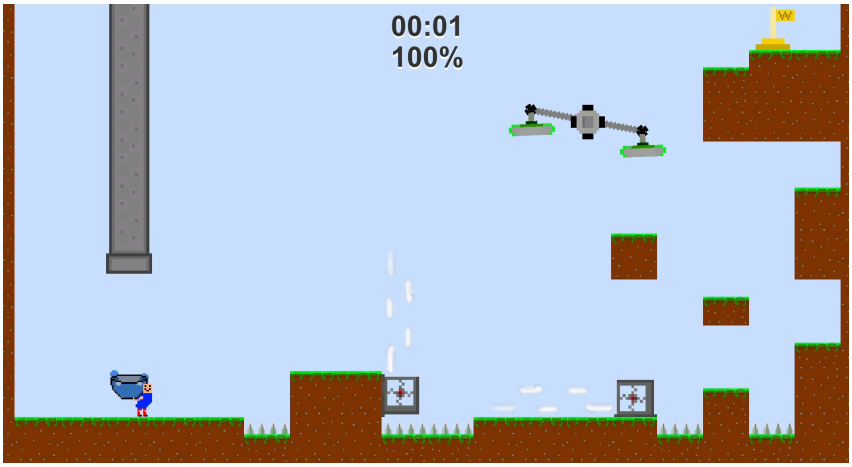
The Leak

A Physics-Based Game Mechanic

- John 'sejohn' Segerstedt

report: 2432 words



Chalmers University of Technology

Dat380 - Technology-driven Experimental Game Design

November 27, 2020

Contents

1 Introduction

This report was written as a part of DAT380 - Technology-driven Experimental Game Design, with the purpose to present and discuss the results and process of the second course assignment on a physics-based game mechanic. This project was the result of one week of ideation and planning, followed by less than two weeks of implementation and presentation.

The chosen physics-based game mechanic was carrying water, see Seciotn 2, within the game prototype 'The Leak'.

This project was a joint effort by John Segerstedt and David Campos Rodríguez.

1.1 Individual Contribution

I, John Segerstedt, were responsible for the following implementations:

- Player character *(user input, sprites, "animations", etc.)*
- Fans *(sprites, "animations", triggers, etc.)*
- Carousel *(sprites, "animations", colliders, etc.)*
- Ground *(sprites, spikes, Tilemaps, etc.)*
- Pipe *(sprites)*
- Audio *(the audiomanager, composing the main theme, etc.)*
- Win & lose conditions *(incl. the GameManager)*
- Player UI

In short, David Campos Rodríguez was responsible for the water simulation, in the form of every script begining with 'Water...', see Section 2, and John Segerstedt was responsible for most of the rest of the prototype, see Section 3.

1.2 Gameplay Patterns

This prototype features the following gameplay patterns, as defined in Björk, S. (2018);

- Evade
- Game Over
- Game State Indicators
- Maneuvering
- Movement
- Obstacles
- Predefined Goals
- Resources
- Scores
- Single-Player Games
- Speedruns

2 Carrying Water

The physics-based game mechanic we have implemented is to have the player transport liquid from one point to another. This concept is combined with a simple 2D platformer prototype, as such a genre requires and encourages rapid player character movement which introduces a new dimension of interesting difficulty. Not just must the player avoid obstacles and reach the end point of the level, they must also make sure that their maneuvering is smooth enough to prevent dropping liquid.

This is the part of the project that David Campos Rodríguez was responsible for.

2.1 Water Collision Implementation

The Water collision detection was written by David Campos Rodríguez and can be found within the 'WaterAlternative' script. It uses the Unity Burst compiler.

Each water particle is affected by multiple attractive and repulsive forces of other nearby objects, see Figure 1, and its attributes, such as position and velocity, are represented by the internal 'WaterState' class.

Then each update tick, each of these WaterState particles are firstly updated in a particle system fashion according to their velocities and gravity. Afterwards, the particles are inserted into a uniform grid of fixed size. Then, the correct density of each water particle and the pressure they experience from surrounding particles are calculated as in Pekke Kuepers (2019).

The next step is to pass through the grid to check for collisions with other external objects, such as the fans or other edge colliders. Collision with edge colliders are the final step in each update tick to prevent the water blobs to "tunnel" through the collider.

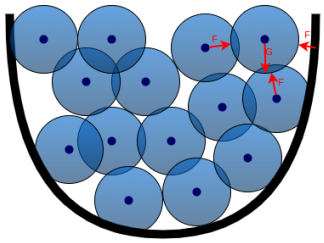


Figure 1: A sketch of how a water particle is affected by surrounding objects.

In the case of a collision with an external collider, represented by an edge collider which is simply an array of positional points, the water particle is firstly pushed outside the collider and then given a correct resulting velocity, see Figure 2.

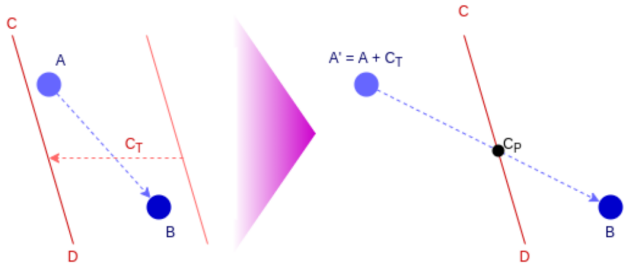


Figure 2: How the state of a water particle is altered to calculate the continuous collision with the movement of the collider.

Additionally, special consideration is needed to be taken with the fan game object, see Section 6, as the water particles are not updated through Unity's native collision detection. Here, the air tunnel is assumed to be axis-aligned, such that its generated edge collider becomes an Axis-Aligned Bounding Box. This interaction can be seen in 3.

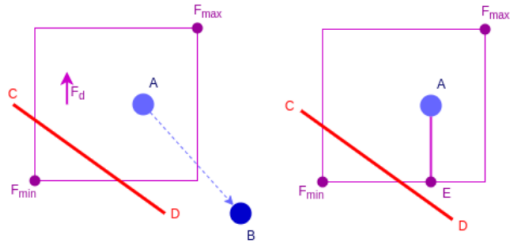


Figure 3: Water particle being updated when inside a fan air tunnel.

3 2D Platformer

This section covers the details regarding the rest of the game implementation, not directly related to the water simulation. This is the part of the project that John Segerstedt was responsible for.

3.1 Player Character

Implementation details regarding the player character Atlas is presented within this section.

The player uses the Model-View-Controller design pattern to separate the responsibilities and allow for further extension.

- *Player* - "model", holds player state and notifies listeners on update
- *PlayerSprites* - "view", in charge of displaying the correct player sprite
- *PlayerMovement* - "controller", manages user input and updates the Player class on turning of the character

3.1.1 Player Input

To control Atlas, the player uses the [A] and [D] keys to apply a sideways force to the left and right, respectively.

All player movement is scaled using Unity's internal frametime counter 'Time.deltaTime' to accurately simulate the player character's movement independent from the current framerate of the game.

The player can also use the [SPACE] key to make Atlas jump. This is simulated by applying a heavy upwards force to the rigidbody of the player character when the event 'Input.GetKeyDown(jumpKey)' is true.

After a jump, the character cannot jump again until it has collided with or landed on another object or the ground. Additionally, new sideways forces as a result of player input gets reduced down to 30% during this airborne time.

3.1.2 Player Sprites

The player character Atlas is composed of multiple self-made 32x32 pixel art sprites, each for the character's various states:

- Moving Left
- Moving Right
- Empty Bowl - Facing Left
- Empty Bowl - Facing Right
- Dead

These can be seen in Figure 4.

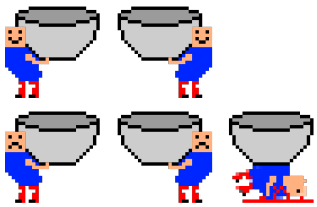


Figure 4: All five sprites for the player character Atlas.

3.2 Level Design

The level implementation utilized Unity's native Tilemap solution to facilitate easy design and redesign of a blocky 2D level, see Figure 5. *(All sprites are self-made.)*

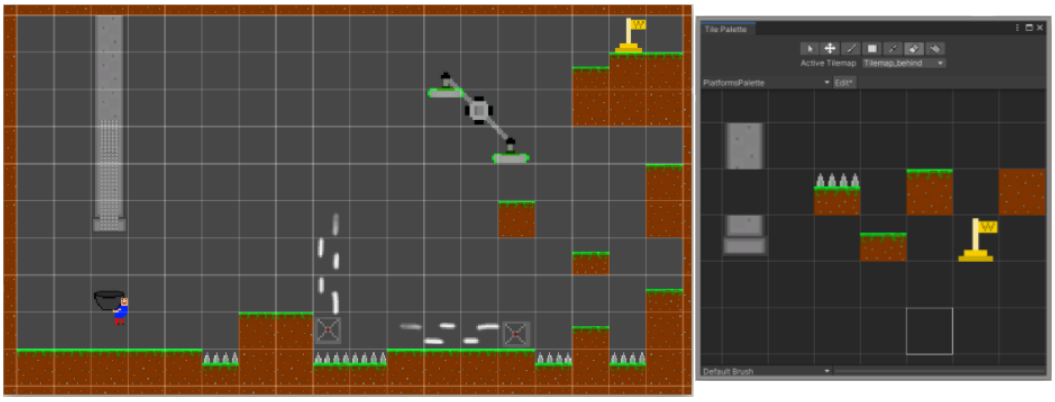


Figure 5: The Grid-Tilemaps implementation and Palette Editor

By using Tilemaps within a Grid, each of the sprites present in the Palette Editor, see Figure 5, scales to the size of a Grid cell and can easily be placed or moved around the 2D scene. Additionally, one can define a collider bounding area for each sprite which then automatically gets merged by adding a 'Tilemap Collider 2D' component to the Tilemap.

Since each Tilemap requires its own collider/trigger configuration, and its own place in the draw order hierarchy, the following Tilemaps instances are used:

- Tilemap: The 'normal' Tilemap for blocks with normal rigidbody colliders. Rendered in front of the player character.
- Tilemap-death: The Tilemap for blocks with the DeathTrigger script that calls '.Die()' on the player after collision with the player character.
- Tilemap-behind: The Tilemap for blocks with colliders that specifically want to be rendered behind the player character.
- Tilemap-no-collider: The Tilemap for blocks without colliders.

3.3 Game Objects

The term 'Game Object' is here referring to objects in the level that the player can directly interact with, or that directly interacts with the player.

During the initial ideation, we wanted two different types of obstacles for the player. One that encourages the player to move onto it and the other being one that encourages the player to avoid it, both potentially causing liquid spill as a result of the rapid movements required. The former inspired the carousel, see Section 3.3.2, and the latter inspired the fan, see Section 3.3.1.

3.3.1 Fan

A fan has two parts; the fan body and the air tunnel.

Figure 6: A Fan 'Game Object' as it appears in-game

Firstly, the fan body is animated by a simple script that constantly transitions between four self-made sprites, see Figure 7. This 'SpriteAnimator' script abstracts the sprite transition functionality in such a way that it can easily be re-used for other purposes.

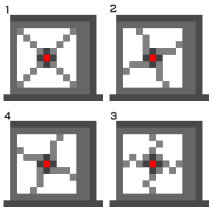


Figure 7: The four sprites animating the fan body.

Secondly, the air tunnel part of the fan also uses the same 'SpriteAnimator' script to transition between multiple self-made air sprites. The air tunnel has its own trigger that adds a force to any collider, in the direction that the fan is pointing, within the trigger area using the 'OnTriggerStay2D(..)' function.

3.3.2 Carousel

The Carousel consists of multiple different parts: the core, the arm, and the hands.

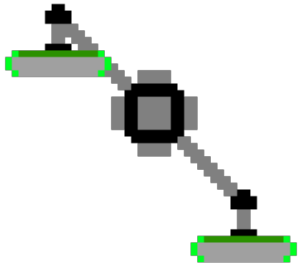


Figure 8: A Carousel 'Game Object' as it appears in-game.

Firstly, the core is the part of the Carousel that is fully fixed in world space.

Secondly, the arm is a long bar that slowly rotates around the core. This is done by the 'RotateObject' script to allow re-use of the rotation code for other object instances in need of rotation.

Thirdly, each of the two hands are connected to the one of the end points of the arm using the component 'Hinge Joint 2D'. This component allows the hands to dangle freely around the Z-axis, while fixed to the arm, as a result of external forces. Additionally, as the hands are child objects to the arm, they share its slow rotation around the core.

Out of these three parts, only the hands have colliders that the player can interact with. To highlight this to the user, the hands have a green highlight,

3.3.3 Win Flag

The Win Flag is a simple object containing only a trigger. When the player enters the trigger area, the Win Flag calls 'SetLevelComplete()' on the 'GameManager' script.

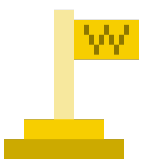


Figure 9: A Win Flag 'Game Object' as it appears in-game.

3.4 Managers

These are singleton classes that have sole responsible for, and acts as the interface to, a specific part of the game "engine".

3.4.1 GameManager

The GameManager script is responsible for keeping track of the game state, and correctly trigger events on changes of this state. If a future implementation of the project has multiple scenes, the GameManager would be responsible for transitions between them.

Additionally, a clear future improvement to the GameManager script would be implementing the Observer pattern to decouple OnGameWin or OnGameLoss events from the manager itself as there would probably be more listeners for these events in a larger project.

3.4.2 AudioManager

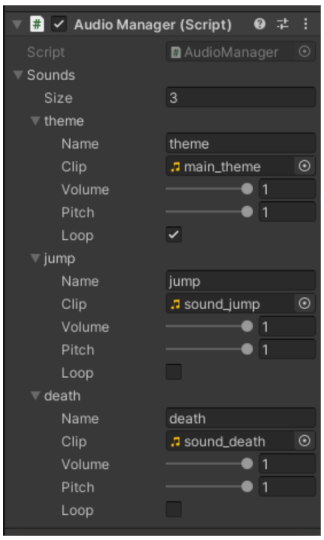


Figure 10: The AudioManager script as it appears in the Unity editor.

The AudioManager script is responsible for maintaining all audio clips that is able to be played by the program.

By abstracting all clip management to a single class, future changes and improvements to these clips becomes easier. As an example, if one were to introduce different AudioMixers and audio levels, i.e. MusicAudio, VoiceAudio, SFXAudio, this is

much more easily done when one's implementation already has aggregated all audio control to a single script.

The main theme and the death sound effects were self made while the jumping sound effect was made by dklon (2013).

The AudioManager script was based off an implementation guide to audio in Unity made by Brackeys (2017).

3.5 UI

Within the game, there are four states to the UI, as can be seen in Figure 11;

- Normal - The startwatch is counting upwards.
- Death - The startwatch has stopped and a message that encourages the player to restart the game has appeared.
- Empty Bowl - The startwatch has stopped, the water in the bowl percentage counter has turned red, and a message that encourages the player to restart the game has appeared.
- Win - The startwatch has stopped and a message appears that both congratulates the player and shows them their score.

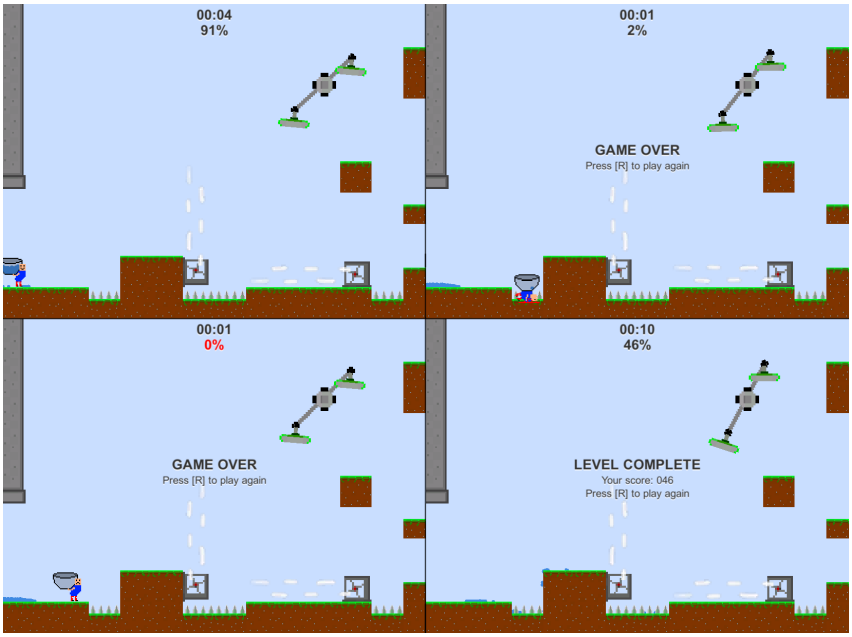


Figure 11: The four different UI states as they appear in-game.

3.6 Win & Loss Conditions

The only current Win Condition in the game is for the player to reach the Win Flag. However, there exists two different Loss Conditions.

The first loss condition is triggered when the player's '.Die()' function is called. Currently, this only occurs as a result of the player character colliding with objects within the Tilemap-Death tilemap, such as the spikes. When this loss condition occurs, the active sprite of the player character's sprite renderer is set to the dead sprite.

The second loss condition occurs when the player's bowl gets emptied of water. This is kept track off in a script called 'WaterCounter'. When this loss condition occurs, the active sprite of the player character's sprite renderer is set to one of the frowning sprites and the water percentage counter, see Figure 11, changes color to red.

When either a win or loss condition occures, the 'gameAlive' boolean within the GameManager is set to false, the player input system is disabled, and a re-start UI appears to the player.

4 Discussion

This section aims to present relevant discussions and potential future improvements to various aspects of the prototype.

4.1 Performance & Stability

The current prototype of the game features 150 water particles and runs at 500-600 fps on a desktop computer with an i5-4460 processor, 16GB DDR3 RAM, and an r9 390 graphics card. For other number of particles, see the list below:

- 150 water particles - 600 fps
- 500 water particles - 400 fps
- 1'000 water particles - 250 fps
- 5'000 water particles - 40fps

The current uniform grid implementation for collision detection allows for the addition of multiple thousand water particles before the frame rate reaches unacceptable levels. However, it is very memory inefficient.

Also, the collision grid implementation is currently only for calculating intra-water forces, which means that every water particle needs to check for collision against the fans during each collider pass.

As far as stability goes, the prototype appears to not experience any visible physical glitches or other errors during any part of normal gameplay as a result of the water simulation.

4.2 Process

Following is the overall structure of the project:

- Week 1 - Ideation and project proposal writing
- Week 2 - Main implementation phase
- Week 3 - Work merging and finalizing & presentation and report writing

The overall work division has worked very well between the two collaborators. There was a clear partitioning of work that well split up the two different implementation

areas, allowing for remote work in parallel with little to no merge conflicts or similar issues.

Continuous intra-group communication was maintained throughout the process and regular coordination meetings were had.

4.3 Future Improvements

The first area of potential improvement is the water simulation. Currently, the particles are allowed to be squished together very heavily when affected by strong external forces, before returning to a resting spread out state. This could potentially be resolved by allowing for more solver iterations.

Also, the collision between particles and game objects with 'environmental effects', such as the air flow of a fan, can be further improved. The current implementation requires completely axis-aligned fan colliders and checks for collision between all water particles and each collider each collider pass as the fans are not added to any collider grid.

Additionally, the visual fidelity of the water particles can be further improved. Either by implementing textures and/or by allowing adjacent water particles to be rendered as a continuous pool of liquid.

The second area of potential improvement is in the 2D platformer implementation. The current implementation requires a specific Tilemap for each type of trigger, which makes for cumbersome level design where one needs to continuously switch and keep track of the current selected Tilemap.

Additionally, when it comes to the animations of the player character, turning is instant. This means that the different colliders for the character are able to instantly appear and dissapear, which may cause them to clip into other objects. This occurs quite rarely but is still a clear area in which the prototype can be improved.

When it comes to the gameplay experience, there is a strong need for more levels. The current prototype can be 'finished' in just a few seconds. Additionally, more interesting level design could be had by implementing more different game objects.

Another initial idea that was scrapped as a result of the time constraint was the possibility for the character to lean depending on the inclination of the ground beneath it.

5 Conclusion

In conclusion, this project was a fun and interesting endeavour that thought me certain aspects of Unity game development that I previously had not encountered, such as joints and tilemaps.

It would have been very interesting to see how far this project could have gone with more development time. It could potentially become a neat casual phone game.

References

Björk, S. (2018). *Patterns*. Accessed on 18 November 2019. URL: http://virt10.itu.chalmers.se/index.php/Category:Patterns.

Brackeys (2017). *Introduction to AUDIO in Unity*. Accessed on 26 November 2020. URL: https://youtu.be/6OT43pvUyfY.

dklon (2013). *Platformer Jumping Sounds*. Accessed on 26 November 2020. URL: https://opengameart.org/content/platformer-jumping-sounds.

Pekke Kuepers (2019). *Simulating blobs of fluid*. Accessed on 26 November 2020. URL: https://peeke.nl/simulating-blobs-of-fluid?utm_campaign=bizarrodevs&utm_medium=web&utm_source=BizarroDevs_55.