How to Make a 2D Soft-Body Platformer With Position Based Dynamics

Zacharie Sciamma
Yale University
New Haven, Connecticut, USA
zacharie.sciamma@yale.edu

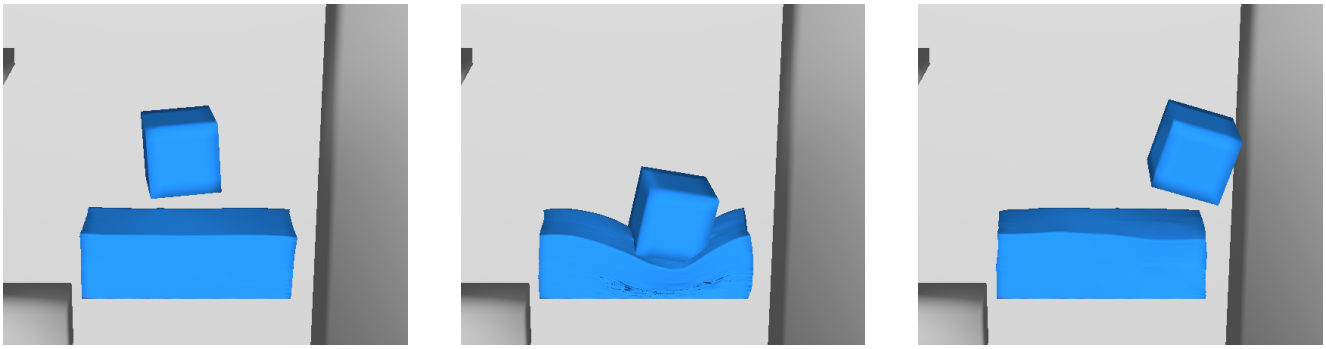


Figure 1: A deformable platform in action

ABSTRACT

The hobbyist game developer community has often pushed the limits of platform games. One popular feature in these games is soft-body dynamics, since it adds interest and greatly extends the range of possible motion. The PositionBasedDynamics library [2] provides a fast and accurate implementation of soft bodies, along with several other physically-based features, but there are currently few guides to help amateur developers learn to use it. In this paper, we describe a method for building a game engine by making small, modular extensions to the PositionBasedDynamics library. Beyond demonstrating how to build the engine, the aim of the paper is to show that this existing soft-body implementation can be used to make fun platformers. The result is a simple, enjoyable physics puzzle game level which uses deformable objects to present interesting challenges to the player. The hobbyist game developer community has always used the tools available to them to make an incredible range of wonderfully innovative games — with this paper, we aim to make one more tool accessible to them.

KEYWORDS

video game engines, physically-based simulation, soft-body physics, position based dynamics

1 INTRODUCTION

This paper presents a game engine built by extending PositionBasedDynamics (Version 2.0.1) [2]. This library is Jan Bender's implementation of position based dynamics, a technique for physically-based simulation — first described by Müller et al. in 2007 [13] — that is both controllable and stable [13]. Bender's library, which is partly based on [3], is thus ideal for use as a physics engine for a video game, supporting many physically-based phenomena such as soft bodies and fluids.

This paper aims first and foremost to make use of the PBD library within a video game engine, and thus provide one of the first guides for using the library in a video game context. The secondary aim is to use the library to create a fun, simple physics game: a game engine is of little use if it cannot make a fun game. So, this paper explains how to use the PBD library to make a game engine, focusing on the features needed to make this proof-of-concept game. The demo game is quite characteristic of the 2D platformer genre, and thus the physics techniques implemented here will be useful to many other games.

At the time of writing, it is difficult for hobbyist game developers to start using the PBD library, since it is primarily a research testbed and does not yet provide a comprehensive manual. The PBD library is not yet accessible to many of the hobbyist developers who might put it to good use, and its high quality and range of complex features make it ideal for platformers, and especially physics-based or puzzle-based 2D and 3D games.

Therefore, much of this project was dedicated to the difficult task of reading the code directly in order to gain a working understanding of the library. This paper aims to fill in some of the current gaps in knowledge, providing both a guide to new users and a demonstration of how the library can be used to make fun games. We show how to extend the PBD library to make a range of core features needed by the game engine, such as the main game mechanics and the camera.

Throughout this guide, we will abbreviate the PositionBasedDynamics software [2] strictly as the "PBD library". Any other phrase that contains "PBD" will instead refer to the general position based dynamics technique in [13], and not to Bender's specific implementation of it.

2 VIDEO GAME DESIGN

2.1 Core concept

In order to build a truly promising demo level, which shows the PBD library could be used in a variety of complete games, it is important to have a well-designed long term vision for our game. Let us therefore describe our design for the full game, which is a much wider scope than this smaller project. The game is a 2D platformer: each level is a unique physics puzzle. The player character is a square which can move around the level only by shooting bullets. Every bullet the player shoots propels them in the opposite direction, thanks to the recoil of the cannon. Using only this unusual method of movement, the player must overcome a range of physical obstacles to reach the end of the level.

The difficulty is that the player only has a small number of bullets, and so reaching their goal will often require executing a series of carefully-timed jumps. At first, the levels will be simple, but as the player progresses, new obstacles will be revealed. The bullets will often be needed to complete a level: for example, shooting enemies will provide stronger recoil and speed boosts, and buttons will need to be hit in order to open gates. Thus the the player must time their shots not only to move around but also to set in motion a series of physical events that will propel them to the end of the level.

Once the player has completed a level once, they must attempt to complete it again, using a different path this time to reach the end. The levels will be carefully crafted such that there are always multiple paths from the start point to the end point: thus, the challenge is to use the obstacles given in creative ways to find all the different ways of completing a level. For example, in our demo level in figure 4, the player can either jump up the platforms on the right or bounce up the wall on the left.

2.2 Use of squishy walls

In this game, PBD will be responsible for generating much of the enjoyment. Our demo showcases the use of the "squishy walls" mechanic, which uses soft bodies as obstacles in the level. The recoil mechanic alone is simple, but combined with soft bodies, it results in a wide variety of jumps and bounces which add a layer of depth to the physics puzzles. Due to the complicated nature of the soft bodies, an interaction between the rigid body player character and the squishy walls can have a wide variety of results depending on the angle of approach and the stiffness of the wall. This increases the difficulty and interest of the puzzles, along with providing satisfying visual effects in every bounce.

2.3 Dual purpose design

A key aspect of this game is the idea of “dual purpose design” [21], a concept used in the game Downwell [12]. In Downwell, items and moves often have multiple functions: for example, the main mechanic involves shooting downwards, which both destroys enemies and keeps the character afloat. Because each feature has multiple purposes, the game is elegant, coherent, and interesting. This concept was inspired by Shigeru Miyamoto, game director at Nintendo, who stated: “A good idea is something that does not solve just one single problem, but rather can solve multiple problems at once.” [21, 00:44 - 00:53]

In our game, each jump will have multiple functions: moving the player around, and shooting obstacles to provide boosts or clear the way. One jump can therefore have a wide range of results. Since a small change in the angle of the jump will give the bullet a very different trajectory, some jumps may have many different effects on the world depending on the precise angle. One certain jump could destroy an obstacle blocking the way, and the same jump at a different angle could shoot enemy and so provide a speed boost. The combination of recoil, bullets, and squishy walls means that a small number of features — and importantly, only a single mouse click — will create a large number of possible outcomes. This establishes the foundation for a wide range of unique physics puzzles. The base language of the puzzle is a few expressive features which can be combined endlessly to craft a coherent and satisfying game.

3 LITERATURE REVIEW

3.1 Existing soft-body solutions

In established game engines, such as Unity [20] and Unreal Engine [8], soft bodies usually have effective, well-supported, and well-documented implementations. However, hobbyist game developers often choose to use more low-level game frameworks over these engines, both to have more technical and creative control over their games, and for learning purposes. For this purpose, there are a number of existing solutions.

In C++, one of the most popular physics engines to support soft bodies is Bullet [6]. In the 2D game engine LÖVE [11], the most popular implementation is the loveblobs library [9]. There are a large number of code-first game engines available, and most of the popular engines have some soft-body implementation — but overall, the number of high-quality soft-body options is not overwhelming, particularly when an open-source library is desired in order to make modifications and extensions.

3.2 Existing PBD library documentation

At the time of writing, few resources exist to help new users become familiar with the PBD library. The main resources are the API documentation [1] and research papers describing PBD more generally, such as [3]. The API documentation provides a list of the existing physics functions; it is mostly geared towards informing advanced users of the available features. The research papers usually describe theoretical rather than implementation details. These are useful for more experienced researchers who seek to understand the PBD library’s underlying algorithms, perhaps to modify it or add features. However, for hobbyist game developers, it is difficult to acquire the necessary context in the first place to understand how to get started with the library.

4 PBD LIBRARY ARCHITECTURE

In this section, we outline the architecture of the PBD library, to the extent that is useful for building the game engine. Most of this information was found by reading through the current latest version of the code (Version 2.0.1), and with little discussion with the author of the library. Therefore, some of these descriptions may unintentionally be incorrect or incompatible with the intentions of the library’s author. However, they provide a useful starting

point for understanding the library, and they allow us to build one possible functioning game engine.

4.1 Directory structure

Firstly, it is useful to understand the overall structure of the PBD library. The `/Simulation` directory contains the classes which represent the physics simulation, the simulation world, and the objects and meshes inside it. These classes define and manage the simulation. Broadly, the `/Simulation` code calls code in the `/PositionBasedDynamics` directory, which contains the classes that handle most of the computation, such as integration and constraint solving. The `/data` directory contains all the data required to build specific scenes: 3D models defined in OBJ files, scenes described in JSON files, and shaders in GLSL files. The `/Utils` directory contains miscellaneous utility classes, such as the `SceneLoader` class, which reads a scene from a JSON file, the `OBJLoader` class, which reads a model from an OBJ file, and the `Logger` class, which logs runtime messages.

The PBD library also provides example programs in the `/Demos` directory. Each demo, stored in its own subdirectory, showcases one or a few specific features of the PBD library. For example, the code in `/Demos/BarDemo` creates a simple flexible bar, and the code in `/Demos/FluidDemo` creates an example fluid simulation. These demos share a large amount (several thousand lines) of common code, stored in /Demos/Common. In particular, the class `DemoBase` in `/Demos/Common/DemoBase.cpp` is responsible for managing the running of the simulation, and initializing and calling the rendering module, `MiniGL`.

In order to run the code, all the necessary files are written `/bin` upon compilation. Each executable is a different demo — compiled from code inside a subdirectory of `/Demos` — and most demos do not require any command line arguments to run.

4.2 Key classes

In this section, we describe the classes of the PBD library which will be important in the later engine implementation.

4.2.1 `DemoBase`. This class, which can be found in `/Demos/Common/DemoBase.cpp`, provides most of the code that will be reused between demos. This is mostly creation and running of the simulation. `DemoBase` handles setting the general simulation parameters and provides functions for rendering every different type of 3D object that might be needed in the simulation.

4.2.2 `MiniGL`. This class, which is implemented in the file `/Demo/Visualization/MiniGL.cpp`, is the rendering module for every demo in the PBD library. It handles almost all the OpenGL calls for rendering basic shapes. This allows complete decoupling of the PBD library from the rendering API, which in our case is OpenGL.

4.2.3 `RigidBody`. This class, in `/Simulation/RigidBody.h`, encapsulates the representation of a rigid body in the simulation. It provides functions to directly change physical properties of the rigid body, such as mass, velocity, and position — but these only have the desired effect if used at specific points in the running of the physics loop.

4.2.4 `SceneLoader`. This class, found in `/Utils/SceneLoader`, is responsible for reading information about a simulation from JSON scene files. A user can create a scene file to define the objects in a specific simulation, such as walls and characters, as well as the running parameters of the simulation, such as time step size. `SceneLoader` then extracts the information from the scene file, determining which rigid bodies, soft bodies, and other objects exist in the simulation, and storing these in a format convenient for later instantiation.

4.2.5 `TimeStepController`. This class, which can be found in `/Simulation/TimeStepController.cpp`, is essential to interacting with the physics simulation, since it runs the main loop of the PBD algorithm (described in [3]). It is responsible for updating and iteratively solving the constraints which bind together the particles of each TetModel, and is thus the main point of access for changing the positions of objects and applying forces on them.

4.3 Useful interactions

For rendering, `MiniGL` encapsulates all the low-level OpenGL calls, including simple rendering functions and window creation. DemoBase handles most of the interactions with `MiniGL`, breaking down more complicated classes such as joints and TetModels into the simpler 3D shapes, such as spheres and tetrahedrons, which can be drawn by `MiniGL`.

The program's main loop is `MiniGL::mainLoop()`, which loops until the user exists the program. At each iteration, it first calls an 'idle' function, then draws the frame, then calls a 'scene' function: the idle and scene functions are set by each demo in order to determine what code should be run in between frames. In practice, the idle function usually triggers a step in the physics simulation, by calling `TimeStepController::step()`. The scene function usually calls the `DemoBase::render()` function, which breaks down the objects in the scene as described in order for `MiniGL` to draw them later.

5 USING THE PBD LIBRARY

5.1 Creating a scene

There are several ways to create a custom simulation. The first is to manually instantiate meshes and objects in a simulation object directly in C++. However, this is not ideal for level design, since it is less flexible. More conveniently, the PBD library provides a framework for reading JSON "scene" files and creating scenes from them. The `SceneLoaderDemo`, one of the demos provided with the library, uses this feature. It loads a custom scene provided by the user in a JSON file, allowing them to play and pause the simulation as well as change its key parameters.

The following steps can be followed to create a new scene:

(1) Create a new JSON file in `/data/scenes`
(2) Add basic simulation parameters
(3) Compile the code
(4) Run `SceneLoaderDemo` with the new scene

Step (1) creates the new scene file which will define all the objects in the scene and the running of the simulation. An example is `/data/scenes/NewScene.json`. For step (2), The simulation parameters can be copied from an existing demo, such as

/data/scenes/PileScene.json — the important objects to edit are Name, Simulation (which contains simulation parameters), cameraLookAt, and cameraPosition.

Step (3) is important, because during compilation, all scene files are automatically copied from /data/scenes to the new location /bin/resources/scenes, so that they can be available as input to SceneLoaderDemo. Then, for step (4), an example command to run the new scene (from the /bin directory) could be:
./SceneLoaderDemo resources/scenes/NewScene.json.

5.2 Adding a rigid body

Rigid bodies can be added to the JSON scene file in an array named RigidBodies. Rigid bodies can be created with a wide range of parameters. As an example, the following snippet would instantiate a single cube rigid body in a scene file:

Listing 1: Creating a single rigid body

```
"RigidBodies": [
   {
       "id": 1,
       "comment": "player",
       "geometryFile": "../models/cube.obj",
       "isDynamic": 1,
       "density": 300,
       "translation": [0,6,0],
       "rotationAxis": [1, 0, 0],
       "rotationAngle": 0.0,
       "scale": [2,2,2],
       "restitution": 0.6,
       "friction": 0.3,
       "collisionObjectType": 2,
       "collisionObjectScale": [2,2,2]
   }
]
```

In particular, the isDynamic parameter defines whether the object can move. Static objects are given mass 0 by the physics engine, which prevents any forces from acting on them, thus freezing their position. The "comment" parameter is only for personal use — it effectively adds a comment in the JSON file, while being ignored by SceneLoader.

The full list of available parameters can be found in SceneLoader, which is responsible for reading all the objects and their parameters. Many parameters also have default values, similarly defined in SceneLoader, which allows the user to omit unneeded parameters in order to keep the scene file simple.

The PBD library provides a few example models. Other models can be created using any 3D modeling software. They must be exported as OBJ files, then placed in the /data/models directory. As with scenes, this directory is copied to /bin/resources during compilation.

5.3 Adding custom soft bodies

Adding soft bodies is similar to adding rigid bodies, but they can contain more complicated parameters. The specific soft bodies we used are called TetModels in the PBD library, since they are constructed from tetrahedral meshes, and we will refer to them as such. Listing 2 is taken from one of the PBD library's example scenes, /data/scenes/ArmadilloCollisionScene.json. It shows how this scene instantiates a deformable armadillo object:

Listing 2: Example TetModel creation from the PBD library

```
"TetModels": [
   {
      "id": 0,
      "nodeFile": "../models/armadillo_4k.node",
      "eleFile": "../models/armadillo_4k.ele",
      "visFile": "../models/armadillo.obj",
      "resolutionSDF": [20,20,20],
      "translation": [0,10,0],
      "rotationAxis": [0, 1, 0],
      "rotationAngle": 1.57,
      "scale": [2,2,2],
      "restitution" : 0.1,
      "friction" : 0.3,
      "collisionObjectType": 5,
      "collisionObjectFileName": "",
        "collisionObjectScale": [2,2,2],
      "testMesh": 1
   },
...
]
```

For more complicated TetModels, an important parameter that can also be used is staticParticles, which allows fixing of specific particles in a model. This is essential to the creation of obstacles for a game, since it allows fixing part of an obstacle in place while the rest remains free to deform and interact with other objects. As with static rigid bodies, static particles have their mass set to 0 upon instantiation, which freezes them in place while still forcing the neighboring particles to stay within the TetModel's specific constraints, thus retaining the object's overall shape.

The library provides a few example soft-body models. Custom soft bodies can be created as follows:

(1) Create a model in 3D modelling software
(2) Export to OBJ format for the visualization model
(3) Export again to PLY format
(4) Use TetGen [18] with the PLY as input to create a tetrahedral mesh for the model (outputting ELE and NODE files [19])
(5) Add the model to the JSON scene file using the generated PLY, ELE, and NODE files

For (1), the specifics of creating the model depends on the desired use, but specific techniques for our case are described in section 6.4.1. Once the model is ready, it needs to be exported to two different formats — as seen in listing 2, adding a TetModel to the scene requires three different files. The "visFile" is a simple OBJ file that is exported directly from the modelling software in (2) (noting that, if the software provides an option to export as a triangular mesh, this must be selected).

In (3) - (4), the same model is then used to create a tetrahedral mesh. While there are multiple software options for this, the armadillo model mesh used in listing 2 was made by the PBD library author using TetGen [18]. This was determined from a comment on the last line of /data/models/armadillo_4k.ele, one of the armadillo mesh files. We therefore used TetGen as well. The TetGen

manual [19] provides extensive information for this, but our chosen method was to export PLY from Blender [5], then use this as input to TetGen, outputting ELE and NODE files [19] which are then used as the TetModel's `eleFile` and `nodeFile` parameters in listing 2.

6 IMPLEMENTING THE GAME ENGINE

In order to convert the PBD library into a game engine, it was necessary to add several features. The code for the resulting game engine is available at [17], which is a fork of the original PBD repository and thus builds on top of it.

For reference, every new feature is contained in its own class. The new classes can be found in the following new files:

(1) `PlayerController`, in the file `/Simulation/PlayerController.cpp` (and `.h`)
(2) `Cannon`, in `/Simulation/Cannon.cpp` (and `.h`)
(3) `Thrusters`, in `/Simulation/Thrusters.cpp` (and `.h`)
(4) `Camera`, in `/Demos/Visualization/Camera.cpp` (and `.h`)

Many unnamed small changes were also made in other files to incorporate these new features into the existing code structure.

While reading these implementation details, section 4.2 should be used as a reference guide.

6.1 Scene creation

Although the example programs are called 'Demos', they show that creating the most simple simulation in the PBD library requires several thousand lines of code. Thus, the most effective approach for us was to use an existing demo and adapt it to include the extra features needed for a game engine. We duplicated `/Demos/SceneLoaderDemo/SceneLoaderDemo.cpp`, which reads a scene from a JSON file and creates the meshes and objects described in the scene, and called this copy `ExternalForceDemo.cpp`.

This file required only a few small changes: the main changes were a few calls to our new objects, such as the `Camera` singleton and the `PlayerController` singleton, so that the scene would now also include a camera and a controllable player character. The other features — the cannon and the recoil — were then added as entirely new classes, called only by `PlayerController`, to retain the modularity of the library. Thus, we effectively converted the PBD library into a small game engine, giving the users freedom to activate whichever features they need.

Our structure also retains the important modular features of the original PBD library, such as its being decoupled from the rendering API. While the `PlayerController` class and its resulting shooting and jumping features (`Cannon` and `Thrusters`) were added to the PBD library itself, the necessary functions for finding mouse position and converting between screen and world coordinates are passed in to the class separately (dependency injection). Therefore, the extended game engine can still be used with a completely different rendering module with no changes to the code in `/Simulation`. This is important because, in a full game, the small `MiniGL` module provided for the PBD library demos would not be used — but a more complicated rendering module would still work easily with our new movement features.

6.2 Recoil movement

The recoil movement is the game's main feature, and it is the core design choice which will provide the fun and the challenge of the game. Adding recoil to the player required being able to apply an external force on objects in the simulation. This functionality did not seem to be available in the PBD library, but it is simple to add, since using the sum of all external forces to update velocities of the particles is one of the main steps in the PBD algorithm [3].

When the mouse button is pressed, the event callback makes a call to `PlayerController::calculateRecoil()`. This function calculates the force to be exerted on the player character, based on the current positions of the mouse and the player. These calculations require several conversions between world coordinates and screen coordinates, but the GLU API (Version 1.2) provides the `gluProject()` and `gluUnproject()` functions [4], which make the conversions simple. As described previously, these functions are passed into `PlayerController` in order to keep the library modular. Once calculated, the force cannot be used yet, since there is no guarantee that the mouse callback took place during the physics loop — applying the force at this point may simply be ignored by the physics. Therefore, the force is stored for later use.

In the `TimeStepController::step()` function, we added a line to call `PlayerController::applyRecoil()` immediately before the projection of new positions takes place (as described in [3]). In the physics frame immediately after a mouse press, `applyRecoil` fetches the recoil force which had been calculated earlier and passes it into `Thrusters::applyPropulsion()`. The `Thrusters` class handles the exertion of the force: it fetches the acceleration of the player `RigidBody` object and increases it according to Newton's second law. Since this is done by definition at the correct point in the physics loop, the new acceleration will be taken into account in the next physics solving iteration, as though an external force had been applied.

Essentially, this mechanism registers mouse presses without enacting them immediately. Once the physics loop is ready to consider external forces, it queries the `PlayerController` to allow it to add any forces needed to fulfil the previous mouse press.

6.3 Shooting bullets

The main mechanic also involves shooting bullets to give the impression that the cannon is causing the recoil. However, instantiating new meshes during a simulation is quite difficult, so we took a simpler approach: creating all the bullets needed off-screen at the start of the simulation, then instantly teleporting them into the simulation when needed.

The `Cannon` class is responsible for launching the bullets in this way. In order to agree with the physics loop, the `PlayerController` calls `Cannon::shootBullet()` immediately after calling the earlier `Thrusters::applyPropulsion()` function. The bullets are given their new position and acceleration, before control is returned to the physics loop, which uses this new information in its next iteration.

The bullet's new position is a world distance of 1 away from the player object's centre, in the opposite direction to the recoil force. Similarly, the bullet is given an acceleration proportional to the recoil force, in the opposite direction. Since the player character is a cube with side length 2, the bullet starts inside the cube volume and

gives the impression of being shot out in the direction of the mouse. The PBD library handles this overlap between the two rigid bodies smoothly, and the player sees the cube effectively being propelled backwards from the recoil of shooting the bullet.

6.4 Making squishy walls

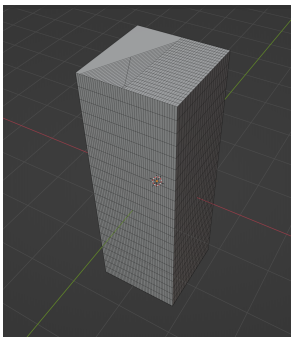
6.4.1 Wall model. The squishy walls mechanic provides much of the game's interest. The first step was to design a 3D model, shown in figure 2. The wall is a cuboid, with a mesh that varies in level of detail throughout its surface. The level of detail determines how fluid the wall will look when hit. Designing the mesh is therefore a balance between adding more detail to create more interesting collisions, and restricting the level of detail to simplify the PBD constraint calculations each frame and thus reduce the amount of lag introduced into the game.

The wall's purpose is that the player can bounce off it. So, in the game the player will interact mainly with one face of the cuboid wall (the "impacted face", the face pointed towards the player in the 3D space). This also means that a second face, the front face, parallel to the game's 'fourth wall', will be in full view of the camera. Thus, these two faces must have detailed meshes; the impacted face so that the player's collisions with the wall can be more interesting and complex, and the front face to avoid jarring visual artifacts caused by the wall's bad topology [15]. The key is that the parts of the mesh which are off screen and never come into contact with the player have almost no details, which will minimize the work done by the PBD algorithm. If the off-screen parts contained many nodes, this would lead to a large amount of time spent solving constraints which in the end have barely any effect either on the visuals or the player interactions.

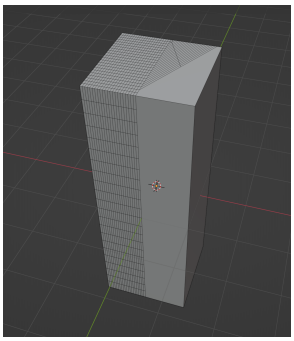
On the other hand, the back face (the face hidden from the player, parallel to the 'fourth wall') must still have a small amount of mesh detail. Even though it is entirely hidden from the camera, it will still receive an impact when the player hits the wall. The impacted face will be crushed, and the back face will support the motion of bouncing back into place. If the back face contains less mesh detail, the impacted face will not be able to properly cave in when impacted, and the player's bouncing off the wall will look unrealistic. So, half of the back wall is finely meshed — the half closest to the player — in order to support the impact of collisions. The other half of the back face is left blank, since it is too far away from the impacted face to affect collisions.

6.4.2 Wall implementation. Once the model was complete, a few more details were required to add it to the game. The wall must be fixed so that it cannot move from its position in the level, but its particles must be free to move in order to deform upon impact. As described in Section 5.3, the PBD library provides a parameter, `staticParticles`, for this purpose. But since the wall model's mesh is complicated, determining the indices of the particles which should be fixed is nontrivial.

We found that the `DemoBase` class provides a `selection()` function which allows a user to highlight particles in a mesh. During the simulation, the user can use the mouse to click and drag over a mesh, and all the particles in that area will be highlighted. We edited this function to print out the indices of the selected particles. Selecting the face of the wall furthest from the impacted face, as



(a) Front view (wall facing player)



(b) Back view (wall hidden from player)

Figure 2: The 3D wall model used in the demo level

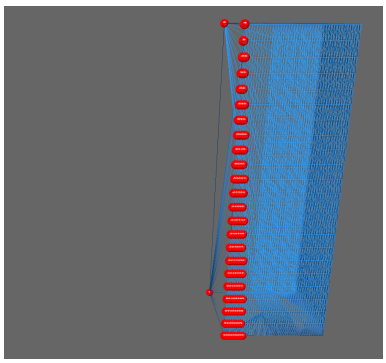


Figure 3: Particle selection in use on a soft-body wall

shown in figure 3, printed out the list of particles which should be static. The `staticParticles` parameter was then set to this list, removing a few exceptions highlighted in the process. The result is that the wall behaves like a soundproof foam panel affixed to a studio wall: part of it is fixed, while the rest is deformable — thus producing squishy walls.

6.5 Making the game 2D

The PBD library seems to support some 2D features, as stated in the repository's `README.md` file, but as we could not find these in our analysis of the code, we instead created a 3D simulation, then used rigid bodies to constrain the objects to the 2D plane. Our demo level contains a back wall and an invisible front wall, both of which are instantiated as regular rigid bodies in the scene.

The PBD library does not support invisible objects, so we added a new parameter to the `SceneLoader` class, which is responsible for reading JSON scene files. This visibility parameter is propagated through to the instantiation of rigid bodies, at which time it is added as a parameter to the `RigidBody` object being created. The rendering is performed by the `DemoBase::render()` function; we added a conditional to check, for every rigid body, whether it has been set as invisible. If so, the `render()` function skips rendering of that object.

The distance between the front wall and the back wall was specific to ensure collisions would still work properly. We chose this gap to be slightly larger than the depth of the player, which means that the player has a small amount of room to move closer to the camera (positive z direction) or further from the camera (negative z direction). In practice, this results in a small amount of rotation as the cube is moving — when the cube changes direction, it also rotates slightly. This is a positive, if unexpected, effect, since it provides depth to the movement and adds character to the otherwise simple cube controlled by the player. The amount of freedom in the z direction was carefully tuned to maximize the effect while keeping the cube's movement effectively constrained to the 2D plane.

6.6 Camera

An important aspect for a platformer is a camera which smoothly follows the player character. It must keep up with the player, preventing the character from leaving the screen while attempting to minimize unnecessary movement, thus preventing any distractions to the player. The inspiration for our camera was the video game Super Mario World [14]. In this game, the camera freezes when the player changes directions, remaining still until the player has moved a short distance in the new direction [10].

Similarly, our camera gives the player some freedom of movement by staying frozen until the player character leaves a circular area in the centre of the screen. Outside this area, the camera moves faster the further the player is from the centre, smoothly increasing its speed up to a maximum in an attempt to catch up with the player.

At every frame, our camera keeps track of the distance vector (d_x, d_y) of the player character relative to the center of the screen. Then, this vector is roughly converted to the order of magnitude of world coordinates by dividing by 100 (such that 100 pixels corresponds to one unit of real-world distance). The total distance is $d = \sqrt{d_x^2 + d_y^2}$. Then, the distance d is scaled using the following operations in sequence:

$$s = \log_{10}(3 \times d - 3.5)$$
$$s = \max(s, 1.5)$$
$$s = \min(s, 4.5)$$

Finally, the camera is moved by c_x in the x direction and c_y in the y direction, where:

$$c_x = -\frac{s}{d} \times d_x$$
$$c_y = -\frac{s}{d} \times d_y$$

Thus, at every frame, the camera attempts to correct its position so that the player is closer to the centre. If the player is within 1.5 world units of the centre of the screen, the camera remains frozen (no movement). If the player is more than 4.5 world units away from the centre, the camera moves its maximum distance (0.01 units). Between these two extremes, the camera moves smoothly: at $d = 1.5$, the log function returns 0, and at $d = 4.5$, the log function returns 1, ensuring smooth transitions between the three stages of movement. These constants were tuned to find the right balance and create a functional camera which catches up fast enough but does not distract the player.

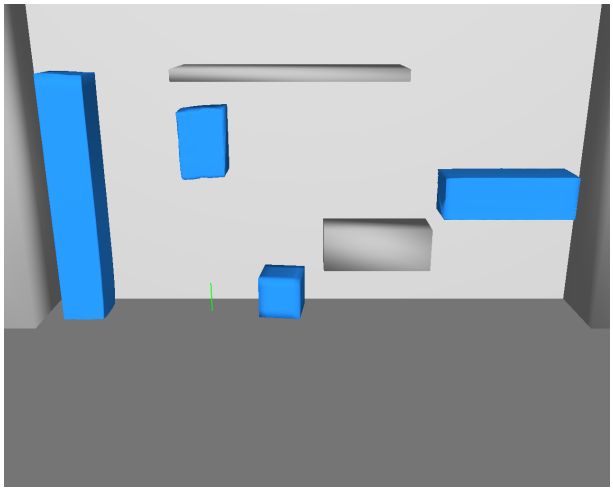


Figure 4: The demo level created using our game engine

7 DEMO LEVEL DESIGN

As described in section 2.3, a common feature of good game design is the reuse of ideas. The implementation of the squishy walls mechanic was general enough to allow us to use the feature in a variety of different ways. The demo level, shown in figure 4, displays two of these, with the squishy walls drawn in blue. The player character (the cube in the center of the screen) must reach the top platform using as many interesting paths as possible — where a path involves hitting at least one obstacle along the way to the goal and using as few bullets as necessary for that path. So, in this level, the player can either step up the platforms on the right, using three bullets, or bounce up the walls on the left, using four bullets. Our gameplay video [16] shows these two solutions, as well as some failed attempts. The two paths are both interesting solutions as they require quite different maneuvers to complete.

An advantage of using the rigorous PBD library as our physics engine is that it allows players to push the game's limits. In fact, one playtester managed to complete the level using only two bullets, surpassing the author's best of three. One particularly fun feature in puzzle games is community high scores. In the future it would be interesting to craft levels such that users have the freedom to find new, unexpected solutions: and the PBD library supports this well.

8 DELIVERABLES

A gameplay video of the level can be found at [16]. The code which generated this demo — our fork of the PBD library [2], extended into a game engine as described, can be found at [17]. Due to the engine's intensive CPU usage, screen-capture tools did not work — so a separate tool was used to capture and save frames (provided in `/Demos/Visualization/FFMPEG_MOVIE.h`, with a few edits), which were then passed into FFmpeg [7] to create the video.

This tool unfortunately cannot capture lag, so a small amount of lag is present in the real game which is not seen in the video.

The deliverables for this project were as follows (abridged from the original project proposal):

(1) Particle system locomotion: the basic recoil mechanic works as expected.
(2) Soft-body dynamics: the 'squishy walls' mechanic works.
(3) (Reach goal) Includes a further tailored mechanic based on soft bodies. Reusing ideas is a fundamental principle of game design, so for this goal, the 'squishy walls' mechanic will be extended to create a more complex and interesting feature for the demo level.
(4) (Reach goal) Includes an additional physics-based game mechanic. This may be based on liquids and/or rigid bodies.

The demo level showcases (1), (2) and (3) — the two required goals and the first reach goal. The recoil mechanic works, and the player can bounce against the walls. The 'further tailored mechanic' is shown in the demo's wall-bounce mechanic: this is a more interesting obstacle which shows that the squishy walls can be used in a variety of interesting ways to present new challenges to the player. In this case, the player must execute three jumps in a row against the squishy walls in order to climb up to the platform.

The second reach goal, (4), would be a necessary future goal for a full version of the game. In a physics-based game, it is important to maintain interest by adding more complex features as the user progresses. This project has done the difficult work of understanding and partly documenting a library which was previously inaccessible to most users, and so including more of the library's features in the game is now much easier — many could be simply added to the JSON scene file, whereas others would require more careful thought to incorporate.

9 FUTURE WORK

For the game engine, the next goal is constraining the speed of the simulation. PBD allows indirect control over the rate of the simulation through control of the time step size and the number of iterations used by the physics solver, but it does not allow direct control over simulation time. It would be ideal to incorporate the physics updates within a gameplay loop, so that the rate of time can be set directly — instead of needing to be tuned as in our project. However, this would be a significant task.

In addition, several useful features could be added to the user interface and game loop, since this project focused primarily on the implementation of the soft bodies. A bullet counter at the top right of the screen could inform the user when they have run out of bullets, and the level should restart automatically after a few seconds. The object of the game is to find several ways to complete a level — so the game should register when a specific surface has been hit by the player, highlighting that object to show success. When the player reaches the goal, the game should indicate that a new path has successfully been found if the player has hit the obstacles in the correct sequence.

Regarding game design, it would be ideal to make more use of the bullets within the level, as specified in the game's original description. This aspect of the game, whereby bullets become an important part of solving the puzzles, would allow for a much wider range of puzzles. These puzzles would be more difficult to design but would result in a much more interesting game, where both skill and careful thought are required to complete a level.

10 CONCLUSION

This paper demonstrates how the PositionBasedDynamics library can be extended into a simple game engine in order to create a fun platformer. Through our experimentation with the PBD library, we discovered many more ideas for features and level designs, from wall bounces, to levels with multiple solutions, to leaderboards. It is hoped that this paper can serve as a guide for users who wish to start using this fantastic library in their own games — and that they will then be able to delve even further into the library's features, revealing more fun and unique game ideas.

ACKNOWLEDGMENTS

Special thanks to Professor Theodore Kim (Yale University) for being my advisor for this project, and for providing invaluable guidance and feedback throughout.

This project was completed for CPSC 490, the Senior Project in Computer Science (Spring 2022), towards the completion of a B.S. degree in Computer Science.

REFERENCES

[1] Jan Bender. 2019. PositionBasedDynamics API Documentation. Retrieved April 30, 2022 from https://www.interactive-graphics.de/PositionBasedDynamics/doc/html/
[2] Jan Bender. 2021. *PositionBasedDynamics (Version 2.0.1).* Retrieved May 4, 2022 from https://github.com/InteractiveComputerGraphics/PositionBasedDynamics
[3] Jan Bender, Matthias Müller, and Miles Macklin. 2015. Position-Based Simulation Methods in Computer Graphics.. In *Eurographics (tutorials).* 8.
[4] Norman Chin, Chris Frazier, Paul Ho, Zicheng Liu, and Kevin P. Smith. 1998. The OpenGL Graphics System Utility Library (Version 1.3). Jon Leech (ed.). Retrieved May 5, 2022 from https://www.khronos.org/registry/OpenGL/specs/gl/glu1.3.pdf
[5] Blender Online Community. 2018. *Blender - a 3D modelling and rendering package.* Blender Foundation, Stichting Blender Foundation, Amsterdam. http://www.blender.org
[6] Erwin Coumans. 2020. *Bullet Physics SDK.* Retrieved May 4, 2022 from https://github.com/bulletphysics/bullet3
[7] FFmpeg developers. 2022. *FFmpeg (version N-106643-g70db14376c).* Retrieved May 5, 2022 from https://www.ffmpeg.org/
[8] Epic Games. 2019. Unreal Engine. Retrieved May 4, 2022 from https://www.unrealengine.com
[9] GitHub user solenum and Phil Hagelberg. 2021. *loveblobs.* Retrieved May 4, 2022 from https://github.com/solenum/loveblobs
[10] Shaun Inman. 2010. Super Mario World Camera Logic Review. Video. Retrieved April 29, 2022 from https://www.youtube.com/watch?v=TCIMPYM0AQg
[11] Love2d forum users Anjo, Bartbes, Rude, Slim, Vrld. 2008. *LÖVE.* Retrieved May 4, 2022 from https://love2d.org/
[12] Moppin, Devolver Digital. 2015. Downwell. Video Game.
[13] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. 2007. Position based dynamics. *Journal of Visual Communication and Image Representation* 18, 2 (2007), 109–118.
[14] Nintendo. 1990. Super Mario World. Video Game.
[15] OReilly.Com. 2014. What is Mesh Topology? Retrieved April 29, 2022 from https://www.oreilly.com/library/view/learning-blender-a/9780133886283/ch07lev1sec1.html
[16] Zacharie Sciamma. 2022. Senior Project - First Simple Puzzle. Video. Retrieved May 4, 2022 from https://www.youtube.com/watch?v=Djj6FgJyC9E&ab_channel=ZacS
[17] Zacharie Sciamma. 2022. *SoftBodyPlatformer.* https://github.com/ZSciamma/SoftBodyPlatformer
[18] Hang Si. 2020. *TetGen.* Research Group Numerical Mathematics and Scientific Computing, Weierstrass Institute for Applied Analysis and Stochastics (WIAS), Berlin, Germany. https://wias-berlin.de/software/index.jsp?id=TetGen&lang=1
[19] Hang Si and A TetGen. 2009. A quality tetrahedral mesh generator and a 3d delaunay triangulator. *Cited on* 61 (2009).

[20] Unity Technologies. 2005. Unity. (2005). Retrieved May 4, 2022 from https://unity.com/

[21] Game Maker's Toolkit. 2016. Downwell's Dual Purpose Design. Video. https://youtu.be/i5C1Uj7jJCg