CPSC 478: StackUp

Lena Ansari, Yichen Cai, and Claire Recamier

Abstract

We set out to create a game called StackUp, where the user positions then drops objects from the top of the 3D game world onto the floor, in an attempt to create the highest possible stack. The score is based on the current height of the stack, which becomes more unstable the higher it grows. The user wins once their stack has reached a certain height threshold. The goal was to create a realistic physical simulation of such a stack, in how differently formed objects might find balance on unstable surfaces and how collisions between such objects might work. We implemented our game world using Three.js and drove the physics of our game using a built-in npm package called cannon-es.

Introduction

StackUp sprang from a combination of all three of our interests. Lena was originally interested in randomness and creating a particle system. Since Claire was interested in physics and particle systems lend themselves to simulations of force-driven kinematics, we considered how to incorporate a particle system into a game that incorporated physics. We were inspired by certain examples shown in class, namely the simulation of hundreds of chairs falling in a physically realistic pile. Upon familiarizing ourselves with the starter code and Three.js, we then began looking into the best way to incorporate physical laws into a simulation, and came upon cannon.

Initially, we thought about programming the physics from scratch using an n-body simulation with gravity. The issue was modeling collisions and whether objects would balance or topple. We considered using a simple approach where if the center of mass was within the x and y limits of the supporting object below, an object would stack. If the center of mass was beyond the dimensions of the supporting object, the object would fall. Thanks to suggestions from our initial pitch, we decided instead to use a physics engine.

Although several other physics engines like bullet and ammo work with Three.js, we chose cannon-es due to its extensive documentation, examples with Three.js, and simpler yet comprehensive features. Cannon-es creates a “physics world” in parallel with the graphics world populated by meshes. The properties of the physics world can be defined arbitrarily with an acceleration gravity vector of any direction/size, an updatable collision matrix, an amount of friction present during collisions, and more. The freedom allowed in the cannon-es physics world means that sometimes physical laws cannot be directly mapped from the real world into cannon-es. Physics parameters must be finetuned through visual confirmation to the scale of the graphics world in order to create a realistic looking simulation. Once initialized, bodies with specific masses, velocities, positions, frictions, and positional or other constraints can be added to the physics world. Each body in the physics world corresponds to a mesh object in the scene with the same shape and size, but cannon-es additionally automates the body according to the defined forces in the scene. To make the effects of the forces visible, mesh objects must be linked with their physics body parallels. This occurs by constantly updating the position of mesh objects using the positions of their parallel physics bodies in the main animation loop of the game.

Alternatively, specifying objects as static instead of dynamic allows objects such as the ground to remain stationary despite the influence of any forces. Finally, the properties of every body in the physics world gets updated in the main animation loop by a simple call to physicsWorld.step(timestep). This moves the bodies according to the forces calculated at call time by a small amount, defined by the timestep variable. Since force vectors change position and magnitude depending on positions of objects, the timestep must remain adequately small for the new force to be recomputed before becoming non representative of the actual physics of the scene.

Once familiar with cannon-es and Three,js and the possibilities of each, we determined our minimum viable product. We wanted to define a physical game that could be controlled by user input. Our basic requirements were to make an object appear at the top of the game world, to position an object in the scene, and to drop the object, all according to user input. We would begin with the simplest object, a cube, and the simplest force, gravity. Already, this minimum viable product required forming a physics world to accurately define gravity and collisions, and tracking user input by projection of 2d mouse coordinates into the 3d camera space. To gamify our simulation, we would add a score based on the height of the user's stack. We also discussed at length the issue of estimating the depth of the object, which was important for the user to recognize in order for them to estimate the best position at which to drop an object. A partial solution that we incorporated into our minimum requirements was to steadily orbit the camera around the center of the scene. Thanks to this camera pan, the user would be able to view the scene from all perspectives and determine the depth of the cube.

Implementation and Methodology

We considered many different approaches to building our game. Initially, we began by building our game on top of the provided starter code, but we rapidly found the starter code an unnecessary complication. The skeleton structure was well-designed for a complex game with moving parts, but separation into many modules made any kind of addition difficult. We struggled to find where in the code we should be adding features and how to import objects across modules. Finally, we switched to a one-module approach built off of the Three.js boilerplate starter code. We found this approach more intuitive and flexible, although it did complicate the meshing of our three codes together.

Upon loading the game, the rendering and camera are initialized, as are all components of the scene - the lighting, background water, sky, ground, and the physics world. We used Three's webGL renderer and perspective camera and leveraged Three's orbit control script for the simplest implementation of a steady camera pan. We designed the lighting with a lens effect layered over a directional light and ambient lighting to look more realistic and create a fully immersive atmosphere. The ocean was created using the built-in Water class in Three.js. We first created a flat plane called waterGeometry, then passed that into the Water constructor along with other configuration properties like texture, color, and distortion. We also updated the water's appearance in our main animation loop using the Three's water class's functionalities for a dynamic animation. Next, we used Three's built-in Sky class to create the sky and to define a sun for directional lighting. We set the style of the sky by modifying the shader parameters.

Initially, we implemented box creation and dropping in a single mouse click. The user simply positioned the mouse anywhere on the screen, and upon a click the box would appear at that point and fall straight down. However, we found that there was an infinite number of points in 3d space that one 2d mouse position could correspond to, and it was impossible to specify the depth at which to drop the box. As a result, we switched to an implementation where pressing the spacebar made a box appear always at the same initial point in world coordinates. The user could then position the box using the arrow keys, which corresponded to a modification in the box's physics body coordinates. The mesh position was updated based on the body position in the main animation loop.

We store references to all (Three.js) box meshes and (cannon-es) box bodies in arrays for easy access. Upon the creation of any box, the box mesh was initialized and pushed to the end of the list. Only upon dropping the box would its body be initialized and pushed to the box body list. This allowed us to delay activating the physics of the scene until the user was ready to drop the box. It also gave us easy access to the current object of interest in the scene, by simply indexing into the last mesh in the box mesh list.

Upon dropping of the box, equivalent to the initialization of the box body, we set the box to have an angular velocity and damping, so the box spins and rotates as it falls. We found angular velocity and damping to add dimensionality to our game by making the stacking of objects less predictable and stable. We also defined a friction between boxes and the ground to kick in upon collision for stabilization.

To help the user predict where the box would fall, we initialized a red dot using Three's point class at the creation time of the scene, but set it to be invisible to the renderer. When a box is created, the dot becomes visible and shifts to indicate the position in the scene at which the box would fall. This is accomplished by casting a ray from the bottom of the object straight down and placing the dot at the point of intersection the shortest distance away. The position of the dot is updated every time the user moves the box to reflect this motion, and set to become invisible once a box begins dropping. Initially, the red dot's position was prone to error, appearing unattached to the surface of any mesh, seemingly out of thin air. This occurred whenever a new box was created before the previous box had settled - the red dot would be initialized to the ray's closest point of intersection at the time of the creation, which was the box in motion's position. Once initialized to that position, the box in motion kept falling, but recasting the ray failed to update the position of the dot, as the closest point of intersection stayed the dot object and not the falling box. To get the dot's position to update correctly, even in the case of positioning before a previous box had landed, the closest intersection had to be filtered by intersection object type. With this fix, the red dot smoothly tracks the box being positioned, even if the point of intersection lies on a box in motion.

Finally, the score is the last element in our game. The score, equivalent to the height of the highest object in the scene, is defined as an HTML element and updated in the javascript script. Every time the score is updated, all the vertices of all the objects in the scene are traversed to find the maximum y position, which determines the score. The game ends when the user has reached a height threshold. Our goal was to have the score only update when all boxes have settled, but detecting when boxes settled was difficult to accomplish. We leveraged cannon

features that detect when an object enters “sleep” mode, defined by its speed delta remaining under a certain threshold for a given amount of time. The score should only be updated once the last box has entered this sleep mode, but must not be updated to a box’s position pre-drop, or the game would be artificially won. We set the score to update after the last box has settled and before the next box is created by the user to prevent this. However, the result is that if a user rushes and initializes a box before the previous one has settled, the score will not be updated until the right conditions are met. The score is replaced with a “you win!” once a height threshold of three stacked boxes is reached.

Results and Next Steps

Although we accomplished more than our minimum viable product, we would have liked to incorporate more effects into the game given more time. One feature to add excitement would have been the ability to stack different kinds of objects. We wanted to explore a whole range of objects, from those built into Three and cannon like polyhedrons and torus, object meshes that we could import such as chairs and tables, and objects with asymmetrical geometries and mass distributions that we defined by hand. We envisioned such objects as having random holes and extensions that could potentially hook onto other objects, dragging down objects from the top of an existing pile. We would also have diversified properties of our objects with textures of different frictions so that objects had unpredictable stackabilities, with some objects more likely to slide off a stack upon receiving an impulse than others. Another potential diversification would have been the rigidity of bodies, as cannon-es allows for rigid and soft bodies. We defined all our bodies as rigid, but we could have modelled pliable objects as particle systems subject to constraints, or cannon-es soft bodies. One example of a more dynamic object would have been a stackable, perhaps square-shaped balloon, that folded and burst under a certain amount of pressure. Given all different kinds of objects, we discussed incorporating random generation of objects, where every creation event generated an object with random properties and a random shape based on a list of options.

Other extensions to increase the unpredictability of the game were to add random gusts of wind that could topple an unstable stack. We also discussed changing the positioning of the floor, perhaps creating a slight tilt or an unstable floor. A simple approach might have been just adding a 3d texture to the floor with displacement mapping, so as to actually physically displace parts of the floor. Other options were to design a beach scene where we modeled the sandy floor using a particle system, which might shift unpredictably under the weight of objects or from gusts of wind. Although we ended up modeling the ground in our scene using water, our ground retained the physical properties of a solid. Another option might have been to incorporate actual fluid dynamics to the water of our scene, with objects partially submerged in water and floating up and down according to wave patterns. Any motion of the surface or material on which a stack was built would have multiplied the difficulty in creating a stable stack.

One graphics extension we considered was refining the red dot, which appears at the predicted point of intersection of a falling object. We considered redefining this as a shadow, perhaps created with a directional light coming from directly above the object. Additionally, we would have preferred a different mechanism to provide the user with different perspectives of the scene. With the constant orbiting of the camera, the user can build up an understanding of the depth of

the object with a full rotation of the camera around the scene, but this approach is an inefficient way for the user to estimate depth and can lead to a frustrating wait. In addition, the camera orbit causes the camera space coordinates to change as the world space coordinates to stay constant, confusing the user's positioning of objects. Sometimes when the user presses the left arrow, the object may move to the left in world space coordinates but this will appear as motion to the right in camera coordinates depending on the camera axes in relation to world axes. The constant rotation also makes the red dot periodically invisible, hidden behind other taller objects based on the camera viewpoint. In these cases the user must wait for the scene to rotate for the red dot to reappear to be able to position the object accurately. A better solution might have been creating two stationary camera viewpoints that the user could switch between, for more control over how to determine depth. An ideal solution would have been total user control over how to position the camera in the scene.

Conclusion

Although there are many extensions we would have liked to add to our game, we were successful in creating an aesthetically-pleasing and dynamic game beyond our minimum viable product. We successfully created a physics world and automated our objects according to physical laws and user input. Collaboratively, we defined the specifics of our game logic, worked through many deployment issues and bugs, divvied up tasks, and finally meshed our individual codes. After exploring different approaches, we were able to implement our StackUp most efficiently by leveraging Three.js and cannon-es.

Contributions

Yichen directed the artistic vision, choosing the water as a backdrop and the combination of directional and ambient lights with lens flaring that creates the mood and beauty of our scene. She also chose the colors and properties of the materials, preferring Lambertian materials for shininess and aesthetics. Lena spearheaded the user input to the game, determining the best way for users to control the creation and the dropping of boxes for maximum user intuitiveness and programming simplicity. This included figuring out how to suspend the physics while the user positioned the box and converting between coordinate systems. Claire implemented the red dot using a raycaster intersections, and the scoring functionality which required a traversal of all objects in the scene to find the highest vertex. Claire also walked the team through all the GitHub setup. All of us separately experimented with cannon and physics, figuring out how to create a physics world and how to drop boxes. Determining the game structure and logic, such as how scores should be calculated, how user input should be specified from different keys, etc was a collective effort on all our parts.

Works Cited

Cannon tutorial: https://sbcode.net/threejs/physics-cannonjs/

Physics tutorial: https://www.youtube.com/watch?v=TPKWohwHrbo

Code framework: https://github.com/WaelYasmina/ThreeBoilerplate

Dropping objects: http://aicdg.com/learn-ammojs/Demo1-Simple%20Scene.html

Sun, sea: https://www.cnblogs.com/dragonir/p/16316217.html

Sleep mode cannon example:
https://github.com/pmndrs/cannon-es/blob/master/examples/sleep.html

Dot rendering in Three js:
https://stackoverflow.com/questions/26297544/three-js-how-to-render-a-simple-white-dot-point-pixel

Raycaster for point of intersection: https://threejs.org/docs/#api/en/core/Raycaster.intersectObject