COS 426 Project Report: Get Down

Abstract

We present an original game, *Get Down*, an endless platformer where the goal is to jump down procedurally-generated platforms and avoid both height damage and getting crushed by the falling ceiling. *Get Down* features realistic physics, aesthetically-pleasing graphics utilizing techniques including simulated toon shading and environment maps, and runs well on a wide variety of computer hardware.

Introduction

You must jump down a series of platforms to outrun a ceiling of spikes which falls faster every second. Fall too hard, and you may not recover. One misstep, and you may never stop falling. How long could you last, and how far could you get down?

In this report, we present an original game, *Get Down*, our take on the endless platformer genre. The goal of *Get Down* is to descend down endlessly generated platforms while escaping a falling ceiling and to get as far down as possible before the game ends in one of three ways: the falling ceiling crushes you, you deplete your health by falling from excess heights, or you fall into the abyss by missing your landing. *Get Down* is uniquely a first-person platformer, but like most endless games like Temple Run, the game becomes progressively harder; in this case, by increasing the speed of the falling ceiling.

To complement the gameplay, we also present a visually attractive look that incorporates popular graphical elements, including environment maps and particles. We implement exclusively dynamically-generated meshes that provide a coherent appearance. Our game is implemented in Three.js [0] and is optimized to run well on older computers (see Graphics/Performance).

Implementation

Gameplay

Using requestAnimationFrame, the game updates all gameplay objects and renders as fast as possible for the player's computer and monitor; for example, players using correctly-configured 60 Hz monitors will play at up to 60 frames per second (FPS), depending on computer hardware [1]. Therefore, many object-updating routines, as well as the controller routine, take into account *delta*, defined as the difference between the current timestamp and previous timestamp,

multiplied by a factor specific to the routine. Because the frame rate differs between different computers and monitors, *delta* ensures that objects update at the same speed.

Figure 1 shows an illustrative side view of the game environment. The player begins at a platform located near the top of the environment. The falling ceiling's speed is influenced by the time elapsed and the distance away from the player, the latter of which keeps the game challenging should the player be far ahead (also known as rubber banding or dynamic difficulty adjustment [2, 4]). The speed is $(0.0001 + 0.0002\sqrt{timestamp} + 0.001\, distance)\, delta$, where *delta*'s factor is 0.1. The square root for *timestamp* is a common practice for endless runner games [3], so as to not increase the difficulty excessively.

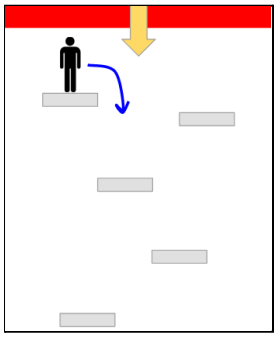


Figure 1: Side view of game environment

Figure 2 shows actual game screenshots taken as of writing. 2a shows the player looking down, 2b looking up.

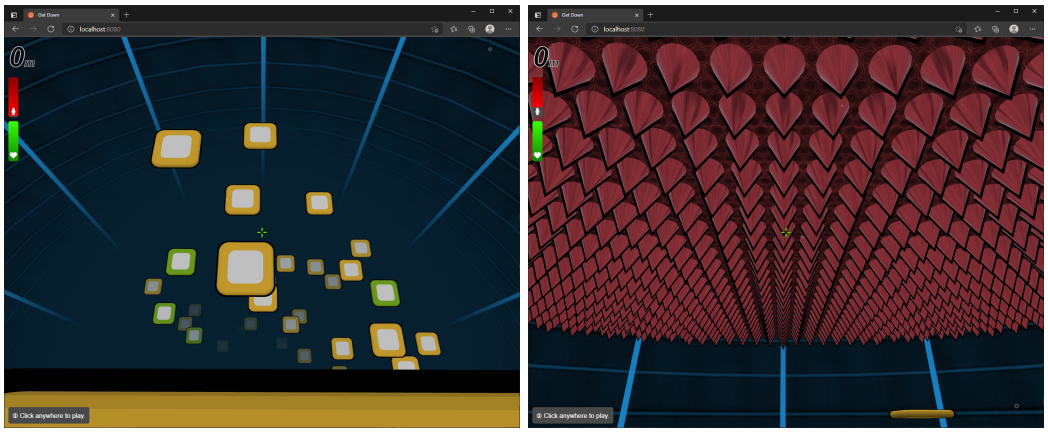


Figure 2a (left) and 2b (right): Actual game screenshots of the player looking down and up

Platforms are generated procedurally. The starting platform is always at (0, 0, 0). All numbers mentioned hereinafter are subject to change. We generate 6 platforms within close proximity to the player. For each of the last 7 generated platforms, we then generate a platform below each of them, with an 80% chance the platform can be reached without height damage, and 20% otherwise; the new platform is located in close proximity in the X and Z axes. We repeat until at least one platform is spawned below a threshold (initially y = -500). The threshold is then moved

downward by 50 Three.js units. When the player's position falls below y = -100, we spawn more platforms until the new threshold is reached. The threshold and trigger positions are then moved downward by 50 units. This way, we guarantee that the player can see all available platforms without pop-in (the fog effects obscures faraway platforms, see Graphics/Platforms) while not having to generate more platforms than necessary. Platforms are removed from memory when they are above the spikes.

We set constraints for platform generation as well. A platform generated below an existing platform cannot be located so that the existing platform overlaps excessively in the X and Z axes; otherwise, it would be difficult to see the new platform from the above existing platform. A new platform cannot be generated beyond the play area (120x120 in the X and Z axes), and a new platform cannot be too close to another; that is, it cannot be spawned in a 20x20x20 "no-go space" box around any existing platform. This collision detection is done using raycasting; see Implementation/Physics/Collision for the general procedure, except we only use one ray here, originating from the center of the new platform pointing to the bottom, and counting the number of times the ray intersects a surface. Odd numbers indicate the new platform's center is inside a "no-go space" box [5]. If a platform breaks any of the constraints, we retry for up to 5 times. If it continues to break constraints, we relax the y-position (the distance from an existing platform to its new platform) and x,z-proximity requirements, then try 5 more times. Lastly, if an existing platform has a new platform that is within safe falling distance, there is a 20% chance the existing platform is warpable (see next paragraph), and is displayed in a different color.

The goal of the game is for the player to make her way down the platforms and avoid being crushed by the falling ceiling. The player has control in the x (A and D keys) and z axis (W and S keys, including in-air, without any slowdown), and can jump if desired (Space bar, or by clicking if the crosshair is not on a warpable platform). If a player cannot find a platform to safely jump to, he can warp to nearby warpable platforms by clicking on it. Height damage will be assessed if the player falls 42 Three.js units or more (15 meters), with the height damage scaled linearly (1% per unit) with the excess fall height. The game is over if the player loses all her health due to height damage or gets crushed by the ceiling. The game is also automatically over when the player is in free-fall for a distance greater than the equivalent in total health.

On the left edge of the screen are heads-up display (HUD) elements. The number at the top indicates the distance in meters the player has fallen. Because the player can jump up platforms, this number is not necessarily the height difference from the starting platform to the current position. If the game is over due to fall damage, the final distance fallen is not counted. To convert from Three.js units to meters, we set 5 units = 1.75 m, rounded to the nearest meter. The red bar indicates the height difference between the player and the ceiling. The green bar indicates the player's health. Gradients are based on [6]. The message box at the bottom appears when the game is paused, resumed, and over. All HUD elements are implemented using HTML, CSS, and jQuery. HUD icons use the operating system's built-in Unicode emojis and may not appear properly on non-current operating system versions. We also implemented full-screen "white-out" and "red-out" effects when the player is falling and has taken height damage, respectively. The opacity of both is determined by the distance fallen.

When the player clicks on the game display, the mouse pointer will be locked into the game and used to control the camera. Furthermore, the game will render and proceed as described above. If the player presses the Escape key, he can pause the game; at which point game objects will no longer update, and no further frames are rendered. Clicking on the game display will resume the game and rendering of frames.

Physics

Since we use a first person perspective, the player's position is simply the position of the camera. Furthermore, a three dimensional velocity vector is defined. These vectors use the Three.js vectors. Based on these primitives, the major components of the game's physics are implemented. These include player movement, gravity, collision detection and warping.

Player Movement

The player is moved using keys bound to the forward, backward, left and right motion. For simplicity, let's assume that the player is moving forward by pressing down on the "W" key. As long as the key is pressed, a constant acceleration of 200 units per second^2 will be applied in the forward direction. Additionally, during every timestep t, a constant deceleration of $-\ 10\mathbf{v}_t$ per second is applied, where $\mathbf{v}_t$ is the player's velocity at time t. In the steady state, the player has a constant velocity of 20 units per second:

$$v_{t+dt} = v_t + 200dt - 10v_t dt$$
$$v_{t+dt} = v_t = 20$$

When the player stops pressing down on the forward key, the player quickly and smoothly decelerates into rest, creating a natural feeling of coming to a stop after sprinting. This mechanic is used in most first person perspective video games. The player also has the ability to jump, given they are currently standing atop a platform. This is simply implemented as a sudden increase of 40 units in the y component of the velocity. We use the Three.js example https://github.com/mrdoob/three.js/blob/master/examples/misc_controls_pointerlock.html as a base for our controller code.

Gravity

When the player is not standing atop a platform and after jumping, the player falls due to gravity. Gravity is implemented as a constant acceleration in the negative-y direction. In our implementation, we find that a gravity constant of $9.8 \cdot 10$units per second^2 balances the game difficulty well.

Collision

In order to detect whether the player is currently standing on top of a platform, we implement collision detection. Collision detection is implemented using the Raycaster object provided by

the Three.js library. In particular, we use five raycasters pointing towards the negative-y axis for bottom collision:

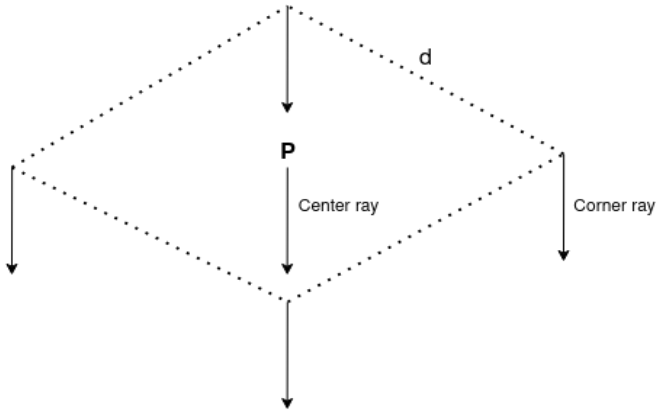


Figure 3: Raycasters used for bottom collision detection

In the diagram above, **P** is the position of the player. The origin of the center ray is **P**, and the other rays originate at the corners of the square of length d centered at **P**. If any of the five rays intersect with a platform within 5 units, the player is marked as having landed on that platform, and gravity stops taking effect on the player. At first, a single raycaster originating at **P** was used, which increases the difficulty of the game, since the player is just a single point in space. Using the setup above gives the player some leeway by simulating a physical body that doesn't fall of a platform, as long as the player is within around $d/2$units from the platform. Although it is possible to directly use a bounding box and detect collision with the box, the described approach was used for ease of implementation. In *Get Down*, the platforms are bigger than the player, so the described approach performs accurately in practice. Also note that each platform is actually made up of collision (invisible), "no-go space" (invisible, see Implementation/Gameplay/Platforms), and display (visible) meshes; collision meshes are co-located with the display meshes, but are slightly larger in the X & Z axes for easier landing.

For top collision detection, the same approach is used with the directions of the rays reversed, and the player collides with a mesh above them if the rays intersect with it within 1 unit. Upon collision, the y-component of the player's velocity is reversed, simulating a perfect bounce. Side collision is not implemented, as it is not a useful feature in *Get Down*.

Warping

In *Get Down*, you are able to "warp" or teleport to specific warpable platforms within a distance threshold, by looking at the platform through the crosshair in the center of the screen, and clicking the left mouse button. This feature is also implemented using a raycaster. The origin of the ray is the camera's position, and the direction of the ray is calculated by applying the rotation of the camera to the initially forward facing vector $(0,\ 0,\ -1)$. As of writing, the distance threshold is 50 units. That is, if the first intersection of the raycaster within 50 units is a warpable platform, the player quickly warps to the position of that platform. The warp is animated using the Tween.js tweening library using quadratic easing.

Graphics

The colors for all in-game objects are based on [7]. Textures are obtained from [8].

Cylindrical walls

Cylindrical walls line the play area. To achieve an infinite cylinder, we drew three cylinders on top of each other, then set triggers between each one. When the player's y-position exceeds a trigger, either the top or bottom cylinder would be removed, a new one would be placed at the bottom or top, and the triggers would be reset. Each cylinder has only a *BackSide* material, so the texture would be visible inside it (see the paragraph after the next for more details). We placed 16 "neon tubes" along the walls of the cylinder, with their x and y positions set as cos(angle) and sin(angle) respectively, where angle is spread evenly between 0 and 2π.

Platforms

To generate rounded platforms, we first draw a rounded square (Three.js *Shape*) with a rounded square hole, then use *ExtrudeBufferGeometry* to form three-dimensional platforms with bevels [11]. To keep the polygon count reasonable while providing a toon shading-like effect without using Three.js *MeshToonMaterial*, the number of bevel segments is set to 4. Then, we use a simple *BoxGeometry* to fill in the center of the platform. To obscure faraway platforms and make it easier to judge platform distances, we implemented a fog effect which affects the shading for all objects 100+ units away from the camera, progressive becoming stronger up until 250 units, where objects are no longer visible due to identical color as the background.

To make the platforms stand out among each other, we implemented a black border around each platform, visible from all angles through a technique best described as the inverse of back-face culling, wherein faces pointing away from the camera view are not rendered (in our case, faces pointing away from the camera are actually rendered, by flipping the normal vectors of the faces; this is handled automatically by Three.js) [9, 10]. To achieve this look, we added a mesh with the same geometry as the platform, but scaled slightly larger. This new mesh envelops the original platform. Further, this new mesh has only a *BackSide* material, so we can only see its backside, behind the platform relative to the camera.

When a platform gets too close to the ceiling in terms of y-position, we implemented a collapsing effect, where we draw nine rounded cubes with appropriate material colors, assign a falling speed and rotation angles to each, and have them fall downwards, rotating and decreasing their opacity every frame.

Ceiling

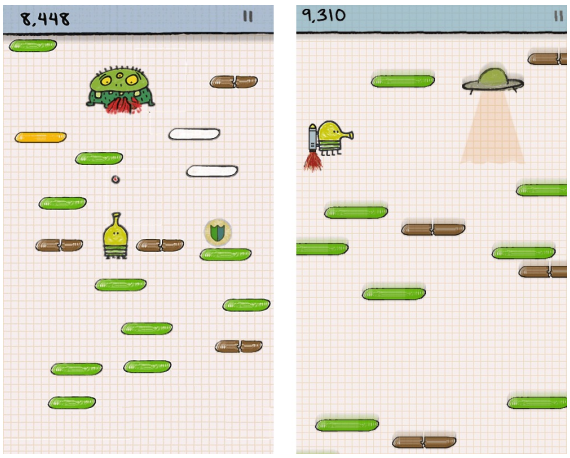
The ceiling is made up of a textured circle with an array of spikes. Spike generation is simply a loop that determines whether the spike will be located on the circle, determined by the distance to the center of the circle. Each spike has 16 polygons, chosen to balance appearance and performance. Spikes use an environment map texture (similar to the texture of the cylinders), which moves the calculated texture depending on the position of the spikes [12]. This technique allows them to appear reflective as the ceiling falls, while making the calculated texture of each spike look uniquely different. Lastly, we implemented the same black borders to the spikes as the platforms.

Particles
We implemented floating particles, which are used to contribute to optical flow as the player moves in the environment, and to provide some animation even when the player is standing still. There are up to 200 particles generated at any given time within close proximity to the player. A particle is assigned a starting position, lifetime, direction, and travel distance at random. For each frame, the particle's position is computed as *starting position* + *direction* * *progress*, where *progress* is determined as the age of the particle divided by its lifetime. The opacity of the particle is calculated as (sin(2 * π * progress - π / 2) + 1) / 2. We use Three.js *Sprite* to implement particles, maintaining their appearance regardless of viewing angle. Image from [13].

Performance
We wrote our object generation routines to be economical on cheaper or older hardware while still being aesthetically pleasing. We tested our game on a 2019 upper-mid-range laptop with two performance modes: standard mode, with a Cinebench R23 score [14] of 3173 points, and power-saving mode, which benchmarks at 2102 points, the latter of which simulates a mid-range ~2016 laptop [15]. Using a 60 Hz monitor and Google Chrome (Windows 10), the game runs at 60 FPS on both modes, with minor stutters on power-saving mode.

Related work

In terms of gameplay, we were inspired by endless runner games such as Doodle Jump and Temple Run. Doodle Jump, another endless platformer, features a small avatar who bounces up a series of platforms, avoiding faulty platforms and enemies with the goal being to get as high as possible. We chose to incorporate the endless platform generation in Doodle Jump while giving the *Get Down* player more control over the jump mechanics, as the jump timing in Doodle Jump is fixed. We also sought to incorporate a progressive increase in difficulty found in games such as Temple Run. In Temple Run, a player runs through endless ancient ruins with increasing speed proportional to time lasted in the game, and we chose to progressively increase the speed at which the ceiling falls in *Get Down*.



Doodle Jump (left) & Temple Run (right)

Stylistically, we drew inspiration from a number of existing video games. We chose to implement the black outline style found in the Borderlands series to give the game a distinctive comic book feel. Though Borderlands uses edge detection built into the Unreal Engine, we used the method described in Implementation/Graphics/Platform to achieve a similar effect in the interest of time We were also inspired by Sega Dreamcast games such as Space Channel 5 in terms of color scheme, polygon count, and rounded edges. The Space Channel 5 color scheme is built on a background of neutrals emphasized by bursts of bright color, as seen in our choice to keep the cylinder a dark blue while accenting the platforms with yellow and making the ceiling red. Our choice to mimic the polygon count and rounded edges seen in Dreamcast games is reflected in our use of simple geometries and rounded platforms.

Borderlands 2 cover

Space Channel 5

The sprite-like particles floating around the cylinder were inspired by the Kokiri Forest in Zelda: Ocarina of Time. We felt these sprites, which float around and shimmer, would add a sense of movement to our game even when the player is standing still.

Note: Refer to Works Cited for all sources of screenshots. The contents of screenshots are properties of their respective copyright owners. All screenshots are presented under Fair Use.

Kokiri Forest

Discussion

We are very pleased with the outcome of our implementation. While we have not performed playtests involving other players, we believe *Get Down* is sufficiently engaging and challenging. We were pleased by the repertoire of built-in Three.js functionality, especially environment maps and the wide range of mesh geometry functions, which meant our code could be written entirely in JavaScript. One aspect of the project we found interesting was trying to balance graphical richness and performance. We wanted to make sure the game runs smoothly on older computers, but we did not want to implement “low-poly” graphics, which we found to be clichéd and overused. Therefore, we tweaked our platform and mesh generation routines to achieve the desired balance. We also considered level-of-detail (LOD), where Three.js automatically swaps in meshes with appropriate detail depending on the distance to the camera, but as of writing, it is not required for good performance.

One graphical element we attempted to implement, but had difficulty and was dropped due to time constraints, was selective bloom or glow effect for the bright tubes on the cylindrical walls. To implement this effect properly (i.e., not just apply shading to the walls), it is necessary to apply post-processing effects, then merge the effect with the original image. However, because the effect both obscures the cylindrical wall, while being obscured by the platforms, it is necessary to write a custom shader [16], which we were not able to complete due to time constraints. Furthermore, we found it difficult to find the right amount of bloom or glow that would work across different display resolutions.

Works Cited

All assets used in our game are licensed under Creative Commons No Rights Reserved (CC0).

Bibliography

0. three.js – JavaScript 3D library [Internet]. threejs.org. Available from: https://threejs.org
1. Window.requestAnimationFrame() - Web APIs | MDN [Internet]. developer.mozilla.org. [cited 2021 May 10]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame
2. Nicolas. “Rubber band AI”: “Rubber band AI refers to… [Internet]. Near Future Laboratory. 2012 [cited 2021 May 10]. Available from: https://blog.nearfuturelaboratory.com/2012/08/23/rubber-band-airubber-band-ai-refers-to-2/
3. arunkarnan. How to balance Speed of the player and length of the Game in an endless runner game? [Internet]. Reddit. 2017 [cited 2021 May 10]. Available from: https://www.reddit.com/r/gamedev/comments/6tu7cx/how_to_balance_speed_of_the_player_and_length_of/dlnkira?utm_source=share&utm_medium=web2x&context=3
4. Dynamic game difficulty balancing [Internet]. Wikipedia. 2020. Available from: https://en.wikipedia.org/wiki/Dynamic_game_difficulty_balancing

5. three.js - threejs raycasting does not work [Internet]. Stack Overflow. [cited 2021 May 10]. Available from: https://stackoverflow.com/questions/33883843/threejs-raycasting-does-not-work#
6. CSS Gradient — Generator, Maker, and Background [Internet]. CSS Gradient. 2017. Available from: https://cssgradient.io
7. Create a palette - Coolors [Internet]. Coolors.co. [cited 2021 May 10]. Available from: https://coolors.co/ff595e-ffca3a-8ac926-1982c4-6a4c93
8. CC0 Textures - Free Public Domain PBR Materials [Internet]. cc0textures.com. Available from: https://cc0textures.com
9. Back-face culling [Internet]. Wikipedia. 2017. Available from: https://en.wikipedia.org/wiki/Back-face_culling
10. javascript - How to add outline on child mesh in three js [Internet]. Stack Overflow. [cited 2021 May 10]. Available from: https://stackoverflow.com/questions/23530879/how-to-add-outline-on-child-mesh-in-three-js
11. Round-edged box [Internet]. three.js forum. 2017 [cited 2021 May 10]. Available from: https://discourse.threejs.org/t/round-edged-box/1402
12. javascript - Three.js: How to add envMap correctly? [Internet]. Stack Overflow. [cited 2021 May 10]. Available from: https://stackoverflow.com/questions/65974012/three-js-how-to-add-envmap-correctly
13. oglsdl. glow circle [Internet]. OpenGameArt.org. 2020 [cited 2021 May 10]. Available from: https://opengameart.org/content/glow-circle
14. Maxon - Evaluate your computer's hardware capabilities | Cinebench… [Internet]. Maxon. [cited 2021 May 10]. Available from: https://www.maxon.net/en/cinebench/
15. Cinebench R23 (Multi-Core) CPU benchmark list [Internet]. www.cpu-monkey.com. [cited 2021 May 10]. Available from: https://www.cpu-monkey.com/en/cpu_benchmark-cinebench_r23_multi_core-16
16. (WebGL) Animated selective glow in Three.js - Blog - (BKcore) Thibaut Despoulain [Internet]. bkcore.com. [cited 2021 May 10]. Available from: http://bkcore.com/blog/3d/webgl-three-js-animated-selective-glow.html

Screenshot sources

Abent, Eric. "Space Channel 5 VR launches later this month." *SlashGear*, 10 February 2020, https://www.slashgear.com/space-channel-5-vr-launches-later-this-month-10609254/. Accessed 10 May 2021.

NY Party Works. "Doodle Jump." *NY Party Works*, 2015, https://nypartyworks.com/product/doodle-jump/. Accessed 10 May 2021.

Takahashi, Dean. "Temple Run crosses a billion downloads. But will it become a global gaming brand?" *Venture Beat*, 4 June 2014, https://venturebeat.com/2014/06/04/temple-run-crosses-a-billion-downloads-but-will-it-become-a-global-gaming-brand/. Accessed 10 May 2021.

Wikipedia. "Borderlands 2." *Wikipedia*, 2019, https://en.wikipedia.org/wiki/Borderlands_2. Accessed 10 May 2021.

Zeldapedia. "Kokiri Forest." *Zeldapedia*, 2014, https://zelda-archive.fandom.com/wiki/Kokiri_Forest. Accessed 10 May 2021.