Game Proposal: Ramster's Revenge

CPSC 427 – Video Game Programming

Team: Ramster Studios

Andrew Yang	#23620222
Davis Song	#29241874
Luke Lu	#25248808
Ning Leng	#11241197
Zach Chernenko	#86974433

Story:

Briefly describe the overall game structure with a possible background story or motivation. Focus on the gameplay elements of the game over the background story.

Our game, *Ramster's Revenge*, is named after the game's protagonist: a genius hamster in a ball who the player will control. *Ramster's Revenge* tells the story of the titular main character's journey breaking out of the lab in which he was made. An evil corporation experimenting on highly intelligent hamsters is in for a rude awakening when Ramster, the smartest of the test subjects escapes from his cage! Shortly after breaking out of the cell, Ramster comes across the HamHook 3000, a high tech grappling hook that allows Ramster to swing across the map and aid in his escape. Living up to his name, Ramster also uses the HamHook to ram into enemies, such as evil lab workers trying to put Ramster back in his cell. Determined and armed, Ramster will stop at nothing to see the outside world and fulfill his dream of living as a normal housepet hamster. Finally, Ramster also has a special catchphrase for when he gets really serious: I'm going HAM!

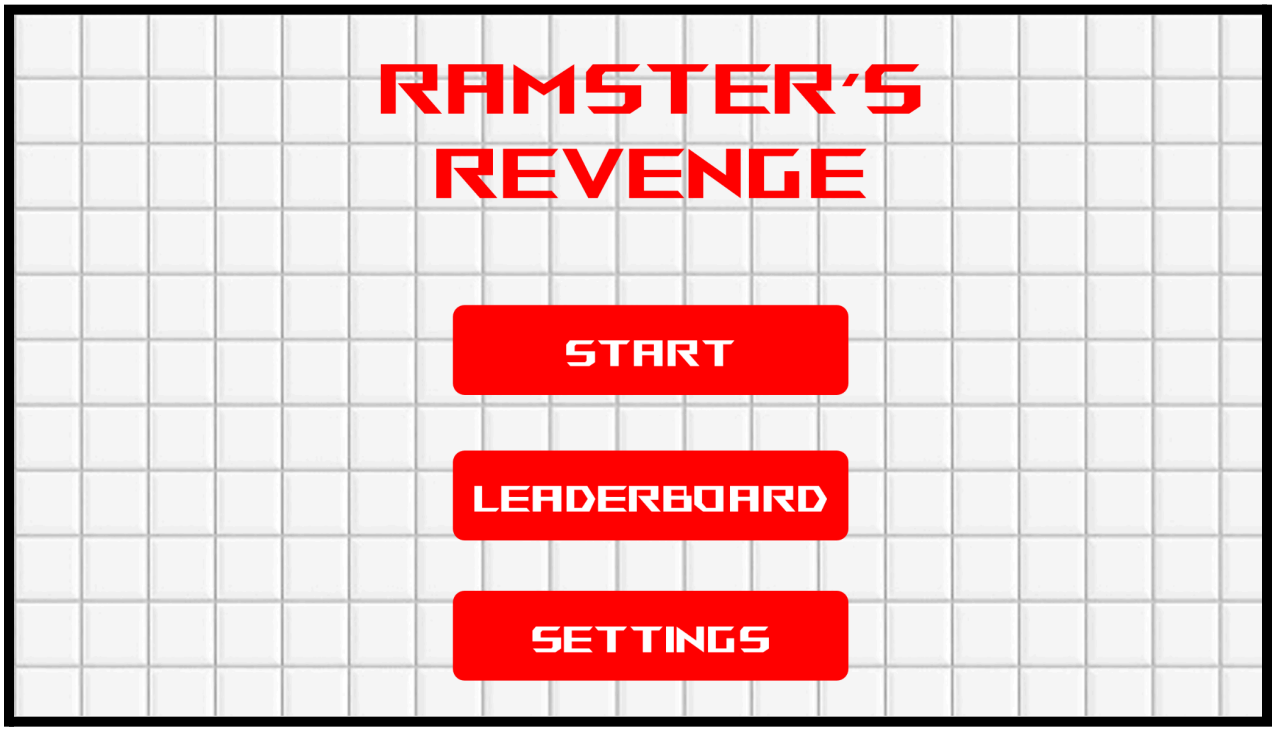
Concept art of Ramster

Concept art for enemy lab worker with jetpack

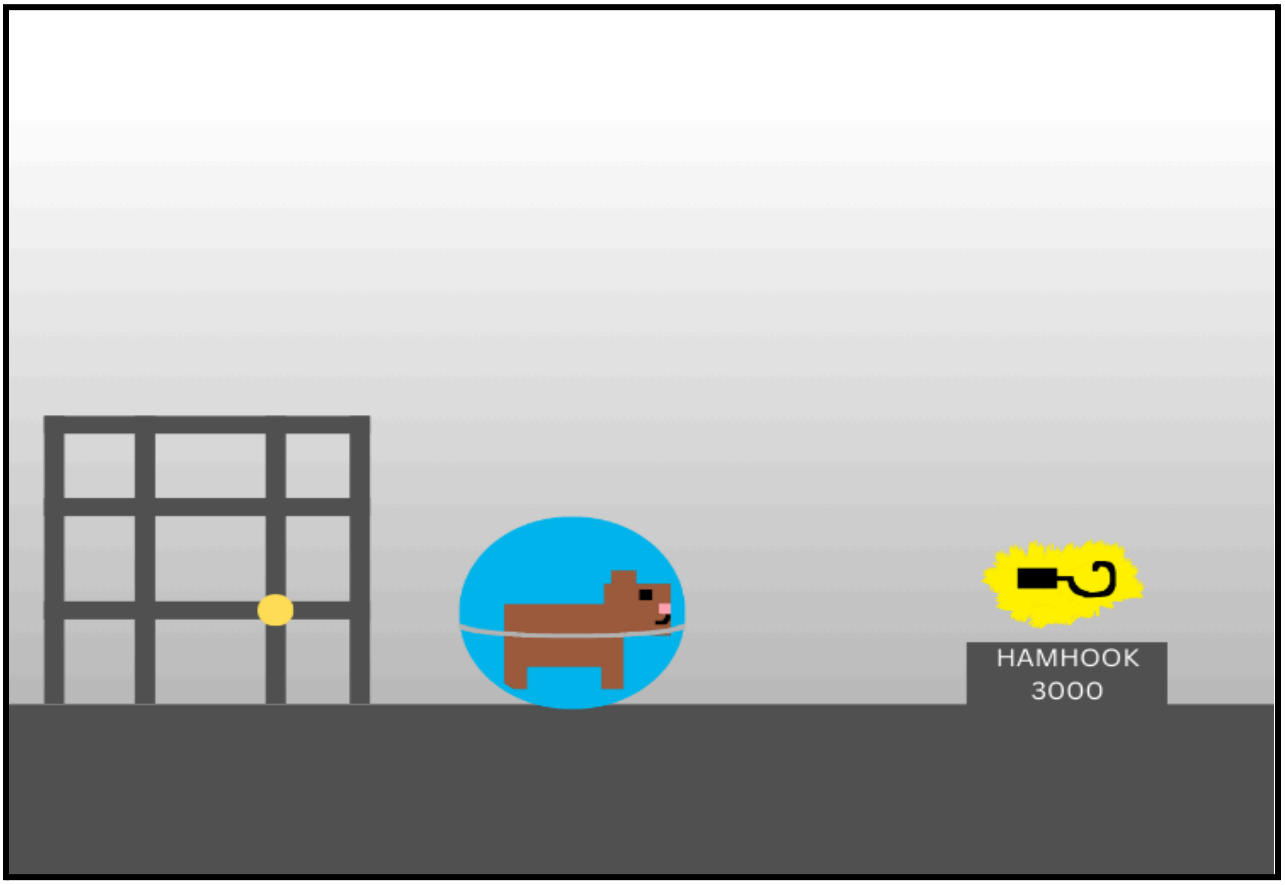
Scenes:

Produce basic, yet descriptive, sketches of the major game states (screens or scenes). These should be consistent with the game design elements, and help you assess the amount of work to be done. These should clearly show how players will interact with the game and what the outcomes of their interactions will be. For example, jumping onto platforms, shooting projectiles, enemy pathfinding or 'seeing' the player. This section is meant to demonstrate how the game will play and feeds into the technical and advanced technical element sections below. If taking inspiration from other games, you can include annotated screenshots that capture the gameplay elements you are planning to copy.

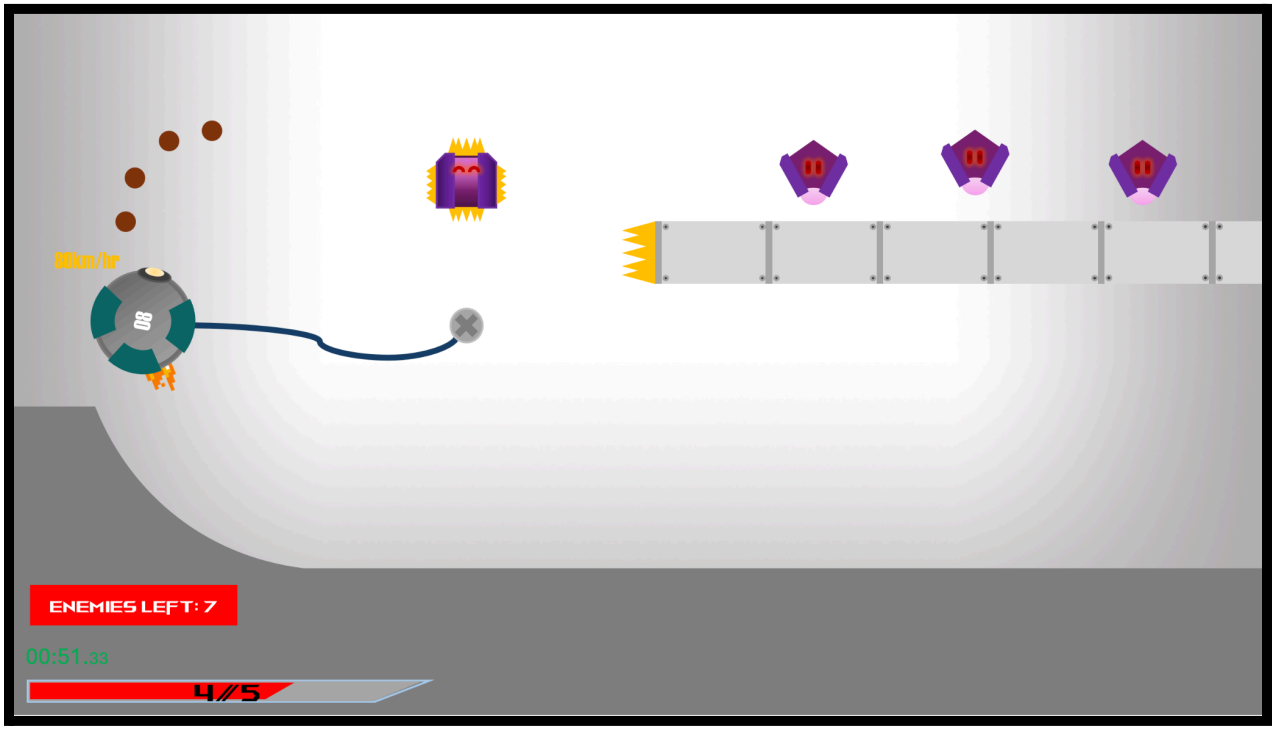
Menu Screen



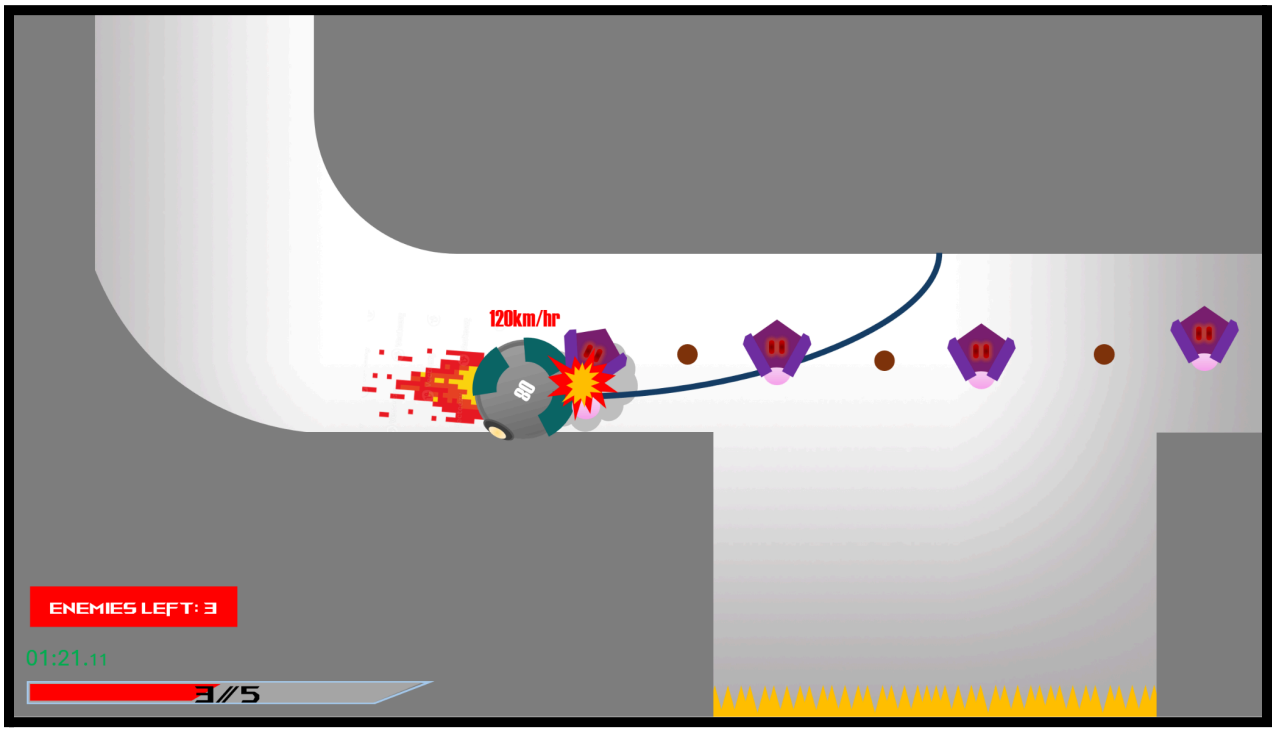
Opening Scene: *after Ramster escapes and receives HamHook*



Gameplay Scene: *using the grapple*



Gameplay Scene: *smashing an enemy*



Stage Complete Scene: *speedrun scoreboard displayed at end of stage*

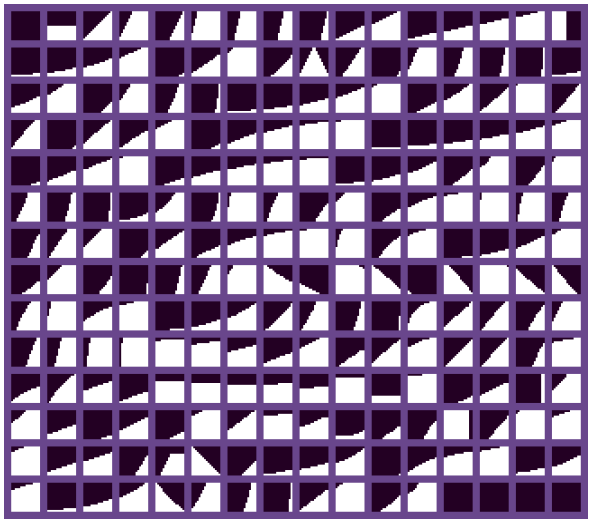
Technical Elements:

Identify how the game satisfies the core technical requirements: rendering; geometric/sprite/other assets; 2D geometry manipulation (transformation, collisions, etc.); gameplay logic/AI, physics.

(1) Graphics:

(1.1) Map System:

As a physics-based game that is centered around grappling and gathering momentum, it is critical that we build maps that are conducive to fun movement. Because of this critical dependency on great level design, we have researched methods that will allow us to efficiently build enjoyable stages. One approach is with tilesets, which is essentially a file containing a number of tile assets that correspond to the actual tiles of the level we intend to render. The data structure representation of tiles would store the angle and height information of their corresponding tiles in the tilesheet. When the player object collides with the tile, it could then look up the angle and height map to adjust its information accordingly. Having access to this data is also important for calculating movement factors such as angular velocity, acceleration, and rebounding behavior.



https://info.sonicretro.org/SPG:Solid_Tiles

For example, the Sonic games used a large tileset of assorted flat ground and curves to account for all the possible types of curves. In this sense, we could define our levels as an array of tiles, such that OpenGL would render the tiles one by one starting from the top left corner of the screen until the bottom right corner.

(1.2) Simple Rendering Effects (Fragment shaders, OpenGL uniforms):

To our best understanding, OpenGL draws each tile in our 2D map by mapping the right part of our tileset texture onto small rectangles (quads), then displaying those quads on the screen. To do this, we store each quad's position and its texture coordinates (which section of the tileset it corresponds to) in buffers that the GPU can use.
We also send extra information, like camera settings or tile slopes, to our shader programs through "uniforms," which are just variables that stay the same for each tile during a draw call. This setup allows us to efficiently render our entire stage in one go while still letting us add special effects (color changes, transparency, or basic lighting) without changing the core drawing process. As detailed in the previous section, by looking up each tile's slope or angle data, we can integrate realistic physics to handle rolling, bouncing, and other movement across curved surfaces.

(2) Artificial Intelligence:

(2.1) Enemy Entities:

Perhaps the most important priority for our gameplay is keeping things dynamic, which means constant movement, preferably FAST. To maintain this atmosphere, we intend on making both the behaviour and difficulty of our enemy entities reward dynamic (and preferably aggressive) gameplay.

We intend on maximizing aggro and ignoring stealth altogether so that enemies will always focus on reaching the player. To emphasize the over-the-top nature of our game, we'll not only make the enemies aggressive, but also large in numbers. One mechanic that forces the player to move fast (or perhaps slowly if they like fighting stuff) is a gradual increase in the spawn rate of enemies with no cap, so the player will eventually be overwhelmed if they stay too long.

We'll try to maintain an orderly chaos, so AI programming should be predictable, such as a shortest-path-algorithm. This way the player will know exactly where the enemies will move, but the movement itself will still be chaotic as we can very much end up with every enemy piling up into a small corner.

(3) Physics:

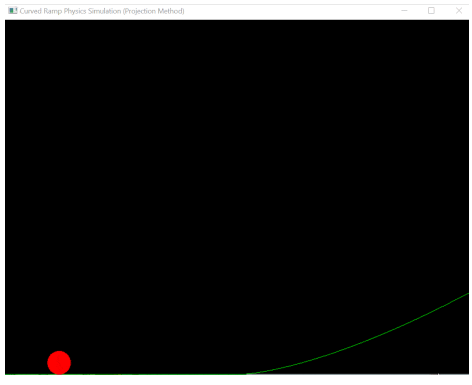
(3.1) Collision Physics:

Collisions themselves must also be over-the-top. Given that hamsters usually don't die from crashing into stuff (or falling from significant heights given their limited terminal velocity), collisions with "normal" map elements might show visual and audio cues, but will not result in a decrease in the player's health bar. We will, however, implement physics surrounding collisions including elasticity and weight/speed difference. There are two main elements on which a collision will be consequential:

- Hitting an enemy entity:
    - When this happens, someone must take damage. We'll make this a function of speed, where the faster entity during the collision will deal damage to the slower entity, and total damage is computed using the difference in velocity. Given that most player-enemy collisions occur between objects of similar weight, there will be lots of stuff sent flying (or losing speed) when collisions occur. Since higher speed results in favorable collisions, the player has all the more reason to move FAST.
- Hitting an interactive map element:
    - Some map elements might damage the player in a high-speed collision, such as spikes (low speed it slows, high speed it impales). Other entities could release shrapnel that damages the enemy. At sufficient speeds, the player would also be able to destroy some map elements that could potentially unveil a buff/nerf to the player.

(3.2) Ramp & Ball Physics:

Because our game revolves around a ball flying around a map, ramps and curves that help the player accelerate are the heart of our game. Therefore, we have already started prototyping ramp and ball physics in C++/OpenGL (description continued on next page).



If viewing in PDF, please visit this URL to view the GIF: https://gifyu.com/image/SeZna

As one can observe from the GIF, the ball should be able to climb and jump up a ramp, yet roll down naturally if the player stops movement input. The geometry

of the ramps and curves should directly influence the acceleration and deceleration of the ball. These will involve dynamic calculations with the game geometry, such as looking up the angle and heightmap of collided tiles. In this sense, we plan to implement complex and realistic ball physics in our game, but may tune the physics to be more in-line with the fast-paced, chaotic nature of our game if necessary. One thought that comes to mind is that we will likely allow for mid-air strafing of the ball (allow use of movement input in the air), which of course is not representative of real life ball physics.

(3.3) Grapple Physics:

Our most complex and main technical feature under the Physics engine is our grappling hook. The player will be able to grapple to any surface and propel themselves forward, building up speed and momentum. The system will operate like a classic physics model, complete with kinematics and gravity. When moving past a certain speed threshold, the model will become fiery. Non-trivial physics properties such as momentum (both linear and angular) and acceleration will be present. Players will be able to indefinitely circle around an anchor point to increase their speed up to a maximum value. They will also have the choice of adjusting their angular velocity by moving in and out along the rope, similar to how a figure skater spins faster when keeping their arms in. As a proven example, see Hammond from Overwatch.

(4) User Interface (UI) & Input/Output (IO):

(4.1) Camera:

The camera should follow the player with a slight delay to ensure smooth transitions, especially at high speeds. This approach helps create a more dynamic feel, preventing sudden jerks when the player accelerates or changes direction. In addition, the user interface (UI) features a sharp, edgy, and sleek red-and-white theme that complements the fast-paced nature of the gameplay.

(4.2) Game Controls:

Players use the WASD keys to roll the ball and strafe in mid-air, while the mouse is used to aim and shoot out the grapple. Pressing the spacebar retracts the grapple, which allows the ball to build speed and perform quick maneuvers. This intuitive control scheme provides precise control over movement and momentum, enhancing the overall sense of speed and fluidity in the game.

(4.3) Audio Feedback + Music:

When the player reaches top speed, a flame indicator and accompanying audio cue signal the heightened momentum. To add to this sensation, wind-up sounds become more pronounced as the ball accelerates, creating a satisfying feedback loop. Upon destroying enemies, the game triggers a small screen shake and a distinct pop sound, further emphasizing the impact of the action. Meanwhile, the background music remains light-hearted and fast-paced—an example can be found here: Sample Track—perfectly matching the energetic, high-speed gameplay.

(5) Software Engineering:

(5.1) Saving/Reloading:

Our game will feature saving and reloading as a technical feature under Software Engineering. After each level, your progress is automatically saved to a JSON file. All metrics for progression will be saved, including best times, pellets collected, and levels completed. Players will be able to select their save file on the main menu screen, and name their file upon creation. If you quit the game while inside of a level, any progress in that specific run of the level will not be saved. Players can also instantly restart the level by pressing a specified keybind. Dying inside of a level will also restart it from the beginning.

(5.2) External Library Integration:

See “Tools” section below.

(6) Quality & User Experience (UX):

(6.1) Assets:
We are planning to have a pixel-art style, which should allow for rapid prototyping. We will start with free assets and tilesets borrowed from the Internet, and hope to eventually draw our own assets for a unified look and feel.

(6.2) Story Elements:
As shown in the scenes, we would like to have an opening scene that depicts Ramster breaking out of his cage in a lab and obtaining his grappling hook. To start, this could be a slideshow of dynamic images, and if time permits, we could animate it for more immersion. One possible option is to find an animation artist on Fiverr who could produce a high-quality animation in exchange for a small fee.

(6.3) Game Balance:

Since our goal is dynamic over-the-top gameplay, we'll buff the player and enemies roughly proportionally so that easier difficulties have less going on and are easier to track, while harder difficulties will be a total gong show. Since the player's main tool is speed, we'll buff that for higher difficulties but introduce more enemies with greater health pools and additional interactive map elements. You can expect easy-mode to be a stroll in a park while hard-mode will more resemble the inside of a tornado. Again, we want the player to be moving, and FAST. So aside from increasing enemies and map elements, higher speed also means higher difficulty to control, which requires greater skill.

Advanced Technical Elements (Stretch Goals):

Describe technical elements you intend to include in the game and prioritize the impact on the gameplay in the event of skipping each of the features and propose an alternative.

(Advanced) Graphics:

Parallax Background (alternative: static backgrounds):

To give the illusion of depth in a 2D game, we could draw multiple background layers that move at different speeds relative to the camera or player. For example, a far-off layer (like trees visible through a lab window) might scroll slowly, while a closer layer (like the wall of the lab) moves a bit faster. Each background layer can be treated similarly to a tile map or a large textured quad, except its position on-screen shifts by a fraction of the camera's movement. By assigning different "scroll speeds" to each layer and rendering them in order (from furthest to nearest), we could create the parallax effect. This technique doesn't change how we set up our shaders or vertex buffers; it only involves adjusting each layer's position and drawing it at the right time in the rendering pipeline.

Procedurally generated levels and/or infinite map (alternative: predefined levels):

With predefined map sections and tiledata, we could create "rooms" that could fit into each other. By chaining several rooms that have compatible connection points together, we could enable procedural generation of levels.

Map Editor (alternative: a text file representation of a map for limited customizability):

A map editor that features an empty map grid and tileset palette such that a developer could place tiles to create a level would be very helpful for rapid level development. This could be extended as a community feature so players can share their created levels.

(Advanced) Quality & User Experience:

Speedrun Timer (alternative: no leaderboard, just post the times):

We want to incentivize our players to go as fast as possible and clear each level in the shortest amount of time. To do this, we will implement a speedrun timer that will start a counter as soon as the player begins the level, and it will end when the level is cleared via killing all enemies and collecting all pellets. Finally, after clearing a level, a leaderboard will be shown including the top times. Your score will be indicated in the leaderboard, with special effects being played if you rank in the top 3 fastest times.

(Advanced) Physics:

Ground Pound (alternatives: n/a):

A stretch goal we have is to introduce a ground pound mechanic. This mechanic will introduce a new way for Ramster to kill enemies. By pressing "q", Ramster will rapidly descend in a straight line before smashing down, and any enemies Ramster collides with during Ground Pound will die. This will create new and interesting ways for players to deal with enemies, incentivizing precise movement with the grappling hook to position Ramster directly above enemies.

Terrain Features & Map Interactions (alternatives: n/a):

We want the map to be more than just trivial obstacles, so we will make many components interactive. Some of our ideas include:

- Speed Tiles: moving over these will increase the player's speed in their direction of motion and applies for as long as they are over the tiles.
- Slow Tiles: like speed tiles, these slow the player down. We might disguise them as bodies of water, bushes, etc.
- Slip Tiles: to make things more chaotic, we'll add tiles that decrease friction, meaning it's harder to change direction over these.
- Punishing obstacles: running into these will damage the player. Think spikes, explosive barrels, etc. Refer to collision physics to see how the player will interact.
- Ramps: Running in a straight line over these will send the player flying. Landing on these will also preserve speed as it gets redirected from vertical to horizontal velocity. Think of hitting a jump while skiing – same idea.

(Advanced) Software Engineering Requirements:

We'll need to implement a couple quality-of-life elements for the player aside from raw over-the-top gameplay. Some ideas include:

- Persistence:
    - Save points: Instantly reset player to start of stage and pause the game upon death or pressing designated reset key.
    - Game progress: Unique players should be able to continue where they left off (not in-map, but in terms of levels) upon re-opening the game.
- Community:
    - Public leaderboard, including stage clear times, speedrun leaderboards, most-enemies-killed, highest-speed-achieved, etc. This will help the best players achieve a sense of accomplishment during their gameplay
- Customizability:
    - Some elements of the game should be customizable, such as volume, brightness, etc.

Devices:

Explain which input devices you plan on supporting and how they map to in-game controls.

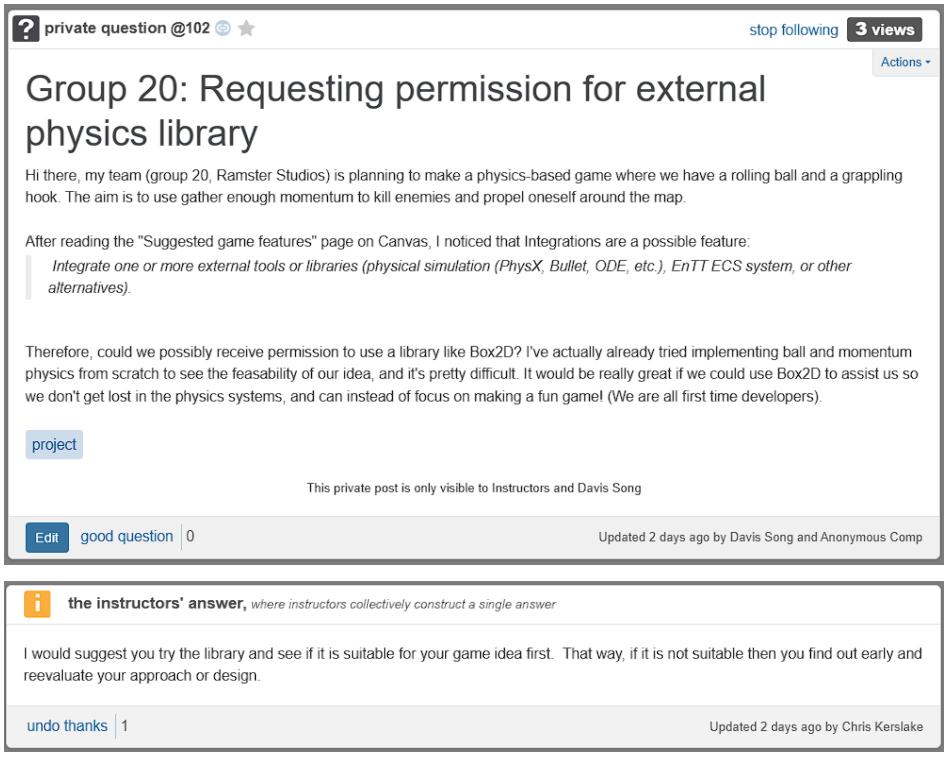
We plan on supporting standard PC input devices, focusing on keyboard and mouse for maximum accessibility and precision. Players will use the WASD keys to roll the ball in any direction and strafe while airborne, allowing mid-flight maneuvers. The mouse is reserved for aiming and firing the grapple at surfaces, enabling quick attachments to the environment. Pressing the spacebar retracts the grapple—helpful for both adjusting momentum and rapidly switching between different points of interest.

Tools:

Specify and motivate the libraries and tools that you plan on using except for C/C++ and OpenGL.

We are considering the possibility of using an external physics library, “Box2D” for our development. Box2D is a free, open-source 2-dimensional physics engine written in C. It offers a suite of useful features, such as Collisions, Translations, Rotations, Springs, and even complex object simulations like Chains. While our current prototyping is with a physics engine built from scratch, using an established library like Box2D could help us build a truly fun game while enjoying the development process ourselves-- as first time game developers, building the physics engine from scratch may be too monumental of a task. Although we have not started formal development yet, we will test out both a physics engine from scratch and Box2D prior and in our milestone 1 development, and see which one fits our game better.

See below for our Piazza question and answer from Professor Kerslake (a soft approval).



private question @102 stop following 3 views

Actions

Group 20: Requesting permission for external physics library

Hi there, my team (group 20, Ramster Studios) is planning to make a physics-based game where we have a rolling ball and a grappling hook. The aim is to use gather enough momentum to kill enemies and propel oneself around the map.

After reading the "Suggested game features" page on Canvas, I noticed that Integrations are a possible feature:

> *Integrate one or more external tools or libraries (physical simulation (PhysX, Bullet, ODE, etc.), EnTT ECS system, or other alternatives).*

Therefore, could we possibly receive permission to use a library like Box2D? I've actually already tried implementing ball and momentum physics from scratch to see the feasability of our idea, and it's pretty difficult. It would be really great if we could use Box2D to assist us so we don't get lost in the physics systems, and can instead of focus on making a fun game! (We are all first time developers).

project

This private post is only visible to Instructors and Davis Song

Edit good question | 0 Updated 2 days ago by Davis Song and Anonymous Comp

the instructors' answer, *where instructors collectively construct a single answer*

I would suggest you try the library and see if it is suitable for your game idea first. That way, if it is not suitable then you find out early and reevaluate your approach or design.

undo thanks | 1 Updated 2 days ago by Chris Kerslake

Team management:

Identify how you will assign and track tasks and describe the internal deadlines and policies you will use to meet the goals of each milestone.

We will use a Google spreadsheet to assign and track tasks. The following policies will be used:

- Tasks will be sorted by type, status, owner, importance, and due date
- Critical tasks will be completed first, then major, then minor
- Once a week (or more) we will meet and triage issues
- Internal deadlines will be decided in these triage meetings and listed under the due date
- If you are reviewing a task, add your name under the Reviewer column

Task List

Task	Type	Status	Owner	Importance	Due date	Reviewer	Notes
Main Story	Story	Not started	Andrew	Critical	1/30/2025		Example
Ground Pound	Stretch Goal	Not started	Andrew	Major	1/31/2025		Notes
Map System	Graphics	In progress	Davis	Minor	2/5/2025		Notes
Parallax Background	Graphics	Not started	Davis	Critical	2/6/2025		Notes
Main AI	AI	Blocked	Luke	Major	2/3/2025		Notes
Grapple Physics	Physics	Completed	Ning	Minor	1/29/2025	Zach	Notes
Camera	UI/IO	Ready for review	Zach	Major	2/1/2025	Andrew	Notes
Save States	Software Engineering	Not started	Zach	Critical	2/1/2025		Notes

Example task assignment (See → Ramster Task List)

Development Plan:

Provide a list of tasks that your team will work on for each of the weekly deadlines. Account for some testing time and potential delays, as well as describing alternative options (plan B). Include all the major features you plan on implementing (no code).

Milestone 1: Skeletal Game

Week 1

- Ball
  - Render in 2D
  - Pressing A rolls the ball left and D rolls the ball right
    - Plan B: If cannot get the rolling physics, implement a simpler sliding mechanic
- Health
  - Displays current health on screen
- Basic Map
  - Flat map with floor and ceiling that handles basic collision with ball (ball does not fall through the map)
  - Map boundary is constrained to screen (cannot move left or right at the edge on your screen)

Week 2

- Basic Grappling Hook
    - Correctly latches onto parts of the map
    - Using mouse to aim grapple and mouse click to shoot it out
    - Pressing A and D when latched on will swing the ball in a circle around latch point
- Basic Enemy
    - Correctly decrements ball's health after ball collides with enemy
    - Static movement (enemy stays still and doesn't move on the map)

Milestone 2: Minimal Playability

Week 1

- Bug fixes
    - Fix any bugs from M0 and M1
- Updated Map
    - Introduce more flat platforms for the ball to move and grapple on
    - Lengthen map boundary, with field of view following the ball as it traverses across the map
    - Introduce spikes on the map that will damage the ball on collision
- Update Enemy
    - Basic AI (constantly moves left and right in a straight line)
- Update Ball
    - Implement collision detection such that a ball moving fast enough will kill an enemy on collision without reducing player health
    - Add fire animation to ball if enough speed gathered to kill enemy on collision
        - Plan B: Omit if not enough time
    - Add music that plays when enough speed is achieved
        - Plan B: Omit if not enough time

Week 2

- Update Grapple hook
    - Refine grappling hook physics to feel smoother/more realistic by adding momentum and gravity
    - Ensure correct collision with platform (rope does not clip through walls)
        - Plan B: Restrict grapple hook to specific marked latch points
- Playability Testing
    - Play test game and take notes on gameplay repeatability, fun, and overall game design

Milestone 3: Playability

Week 1

- Bug fixes
    - Fix bugs from previous milestones, make sure there are no memory leaks, crashes or glitches
- Update map
    - Introduce slopes with ramps that the ball can launch from using the grapple hook
        - Plan B: If slopes prove to be too difficult to implement, use a stepped terrain with springs on each step to mimic original design
- Update Ball
    - Implement the necessary physics for ball movement on the slope, making sure the ball's relative position to the slope matches it's motion
- Persistence
    - Implement a save and load system
        - Plan B: Omit and have players start a new game every time they launch

Week 2

- Playability Testing
    - Continue to play test game, this time with people outside the group, and take notes on gameplay repeatability, fun, and overall game design
- Update Grapple Hook
    - Add the ability to retract the rope, allowing the player to pull the ball towards the latch point
        - Plan B: Omit if not enough time
- Update Enemy
    - Update enemy AI to use a pathfinding algorithm to locate the player
- Speed Timer
    - Implement speedrun timer and leaderboard, tracking time it takes to complete a level

See Milestone 4 on the next page.

Milestone 4: Final Game

Week 1

- Loading Screen
  - Create loading screen to start game, view achievements, and change settings
- Settings Screen
  - Create settings screen for players to change keybinds
- Achievement Screen
  - Create achievement screen for players to view achievements
- Update Map
  - Implement a procedurally generated map allowing for infinite mode
  - Add a parallax background
- Opening Scene
  - Create animation for opening scene where Ramster breaks out of jail and gets grappling hook
    - Plan B: Make it a static scene with no animations and simply some text

Week 2

- Ground Pound
  - Implement ground pound allowing ball to rapidly smash down on the ground
- Final Video
  - Prepare final video showcasing our video game