GAME DEVELOPMENT

JAVASCRIPT & TYPESCRIPT

ARYAN KAUL

PAPER NAME

BOOK.pdf

AUTHOR

Aryan Kaul

WORD COUNT

9834 Words

CHARACTER COUNT

57039 Characters

PAGE COUNT

57 Pages

FILE SIZE

1.8MB

SUBMISSION DATE

Feb 28, 2024 1:02 PM GMT+5:30

REPORT DATE

Feb 28, 2024 1:03 PM GMT+5:30

2% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

- 1% Internet database
- 0% Publications database
- Crossref database
- Crossref Posted Content database
- 1% Submitted Works database

Excluded from Similarity Report

- Bibliographic material
- Quoted material
- Cited material
- Small Matches (Less then 14 words)

TABLE OF CONTENTS

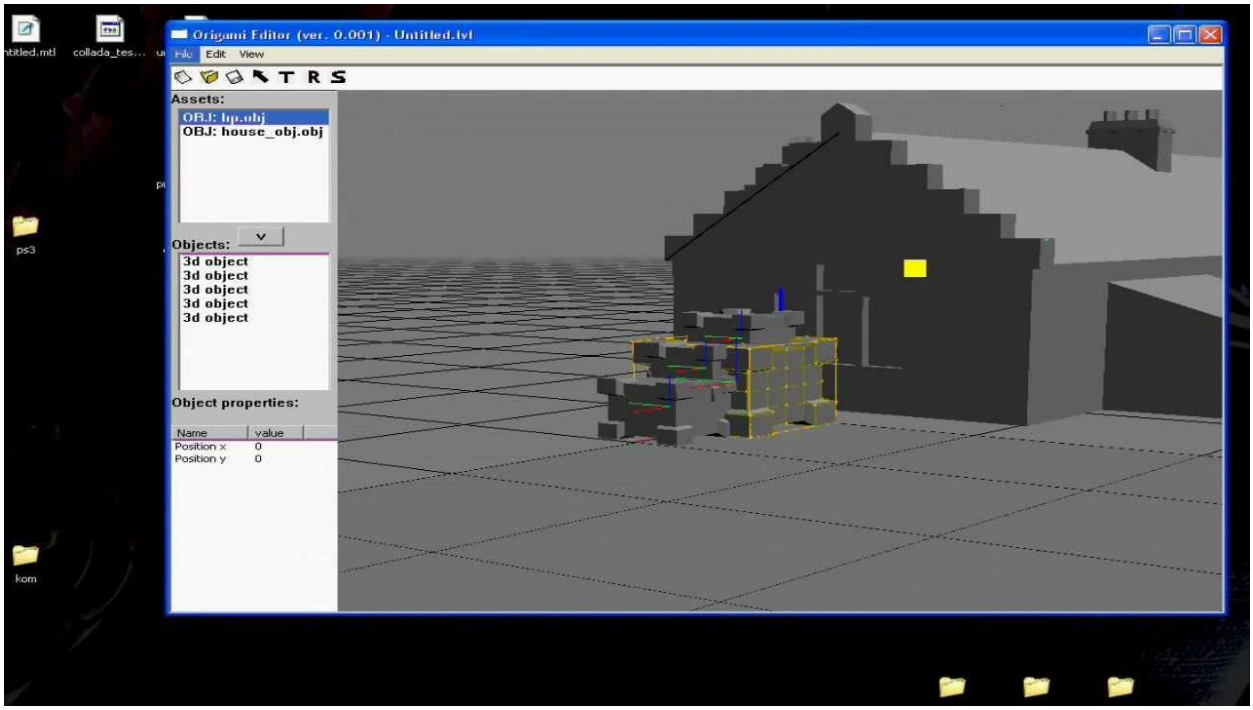
CHAPTER 1: INTRODUCTION OF THE PROJECT

1.1 – INTRODUCTION TO GAME DEVELOPMENT

‘‘Game Development” is the process or art of creating, designing, development and release of a game. It is a multidisciplinary practice that requires technical knowledge and skills to turn game concepts and ideas into reality. It is very important to think about the game mechanics, rewards, player engagement and level design. Game development can be carried out by individuals or by teams in gaming studios. With the help of a variety of game development engines, the process has become more accessible, even to those without extensive coding skills.

A game developer could be a programmer, a sound designer, an artist, a designer, or many other roles available in the industry. To get involved in the Game Development process, you do not need to write code. Artists may create and design assets, while a Developer might focus on programming a health bar. A Tester may get involved to see that the game works as expected.

To resolve problems that game frameworks had, tools like libGDX and OpenGL were developed. They helped game development to be a lot faster and easier, providing lots of pre-made functions and features. However, it was still hard to enter the industry or understand a framework for someone coming from a non-programmer background, a common case in the game development scene.



1.2 – HISTORY OF GAME DEVELOPMENT

The history of video games began in the 1950s and 1960s when computer scientists started creating simple games and simulations on minicomputers and mainframes. Notably, Spacewar!, developed by MIT student hobbyists in 1962, stands as one of the earliest video games displayed on a screen.

In the early 1970s, the first consumer video game hardware emerged. The Magnavox Odyssey became the inaugural home video game console. Simultaneously, the arcade scene witnessed the birth of iconic games like Computer Space and Pong.

As Pong's popularity soared, numerous companies entered the market, leading to cycles of innovation and saturation. These early milestones laid the foundation for the dynamic and ever-evolving video game industry we know today.

Above are some images of Spacewar game and the machine.

The Evolution of Video Games: A Journey Through Time

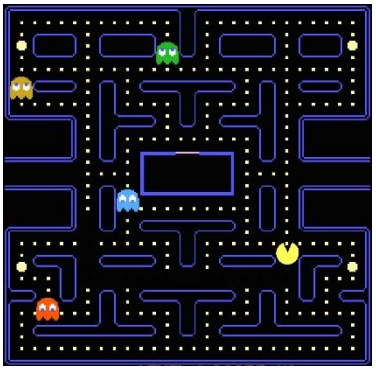
In the mid-1970s, a significant shift occurred in the world of video games. Low-cost programmable microprocessors replaced the earlier discrete transistor–transistor logic circuitry. This technological leap paved the way for several key developments:

1. ROM Cartridge-Based Consoles:
    - The Atari Video Computer System (VCS) emerged as one of the first home consoles to use ROM cartridges.
    - These cartridges allowed players to swap games easily, expanding the console's library.
2. Arcade Golden Age:
    - The 1970s and 1980s witnessed explosive growth in arcade video games.
    - Classics like **Space Invaders** and **Pac-Man** captivated players worldwide.
    - Simultaneously, home consoles gained popularity.
3. The 1983 Video Game Crash:
    - The U.S. market faced a saturation of games, many of poor quality or clones.
    - Inexpensive personal computers and new game types posed competition.
    - Japan's video game industry stepped up to lead the market.
4. Nintendo's Revival:
    - In 1985, Nintendo released the **Nintendo Entertainment System (NES)** in the U.S.
    - The NES revitalized the industry, offering iconic games like **Super Mario Bros.** and **The Legend of Zelda**.
5. Console Wars:
    - The late 1980s and early 1990s saw fierce competition between Nintendo and Sega.
    - Both companies vied for market share, driving innovation and captivating players.
6. Handheld Consoles:
    - The 1990s introduced major handheld consoles.
    - Nintendo's **Game Boy** platform became a cultural phenomenon.

the video game industry's journey from early microprocessors to today's intricate narratives and breathtaking visuals reflects not only technological progress but also the passion, creativity, and resilience of countless developers, artists, and players.

Here are some differences between modern day games and old games –

1. Graphics and Rendering:
    - **Early Games**: Used pixel-based graphics with limited resolution. Sprites were simple and often represented by a grid of pixels.
    - **Modern Games**: Utilize 3D models, shaders, and advanced rendering techniques. High-resolution textures and realistic lighting create immersive visuals.
2. Sound and Audio:
    - **Early Games**: Had basic sound effects generated by simple waveforms. Limited channels for music and sound.
    - **Modern Games**: Feature complex audio engines, dynamic music, and surround sound. Realistic soundscapes enhance gameplay.
3. Physics and Simulation:
    - **Early Games**: Often lacked physics simulations. Movement and collisions were basic.
    - **Modern Games**: Implement physics engines for realistic interactions—gravity, friction, and complex object behaviors.
4. AI and Behavior:
    - **Early Games**: Minimal AI—simple patterns or scripted behaviors.
    - **Modern Games**: Advanced AI algorithms for NPCs (non-player characters). NPCs react dynamically to player actions.
5. Game Engines and Frameworks:
    - **Early Games**: Hand-coded in low-level languages (assembly, BASIC).
    - **Modern Games**: Built using powerful game engines (Unity, Unreal Engine) with high-level scripting languages (C#, C++).
6. Networking and Multiplayer:
    - **Early Games**: Mostly single-player experiences.
    - **Modern Games**: Extensive network code for online multiplayer, matchmaking, and real-time synchronization.
7. Memory and Optimization:
    - **Early Games**: Limited memory—developers optimized every byte.
    - **Modern Games**: Abundant memory allows for detailed assets and complex game worlds.



1.3 – OBJECTIVE OF THE PROJECT

Our Objective for The Project is as follows -

1. Simplicity and Accessibility:
    - **Objective**: Simplify game development for beginners and intermediate developers.
    - **Rationale**: Many aspiring game developers find the initial learning curve daunting. By providing accessible tools, we aim to lower the barrier of entry and encourage more people to create games.
2. Rapid Prototyping and Iteration:
    - **Objective**: Enable rapid prototyping and experimentation.
    - **Rationale**: Game ideas often evolve during development. A framework that allows quick iteration—testing mechanics, visuals, and gameplay—facilitates creativity and innovation.
3. Cross-Platform Compatibility:
    - **Objective**: Develop games that run seamlessly across different platforms.
    - **Rationale**: Players use various devices—desktops, laptops, mobile phones, consoles. A framework that supports cross-platform deployment ensures wider reach and engagement.
4. Efficient Asset Management:
    - **Objective**: Streamline asset handling (sprites, sounds, music, etc.).
    - **Rationale**: Managing assets efficiently saves time and reduces complexity. Developers can focus on game logic rather than file management.
5. Community and Documentation:
    - **Objective**: Foster an active community and provide comprehensive documentation.
    - **Rationale**: A supportive community helps developers troubleshoot issues, share knowledge, and collaborate. Clear documentation ensures developers can utilize the framework effectively.
6. Performance Optimization:
    - **Objective**: Optimize performance without sacrificing ease of use.
    - **Rationale**: Well-optimized games run smoothly, enhancing player experience. Balancing performance and simplicity is crucial.
7. Encourage Creativity and Experimentation:
    - **Objective**: Empower developers to think outside the box.
    - **Rationale**: Game development is an art form. A framework that encourages experimentation—whether through unique mechanics, visual styles, or storytelling—nurtures creativity.

By achieving these objectives, we aim to make game development more accessible, enjoyable, and productive for developers of all skill levels.

1.4 – PROBLEM STATEMENT

Game development is a dynamic and creative field, but it can also be complex and intimidating, especially for beginners. Aspiring game developers often face several challenges.

Our project aims to address these challenges by achieving the following objectives:

1. Simplicity and Accessibility:
    - Develop a framework or utilize existing ones (like **Kaboom.js**) that offers an intuitive syntax and component-based system.
    - Provide clear documentation and tutorials for beginners to create their first games without feeling overwhelmed.
2. Rapid Prototyping and Iteration:
    - Enable developers to quickly experiment with game mechanics, visuals, and interactions.
    - Facilitate iterative design by allowing easy adjustments and refinements during development.
3. Cross-Platform Compatibility:
    - Ensure that games created using the framework can be deployed across various platforms (web browsers, desktop, mobile).
    - Simplify the process of adapting game mechanics to different devices.
4. Efficient Asset Management:
    - Implement tools for organizing and managing game assets.
    - Streamline asset integration, reducing the time spent on file management.
5. Community Support and Collaboration:
    - Foster an active community around the framework.
    - Encourage collaboration, knowledge sharing, and troubleshooting among developers.
6. Performance Optimization without Complexity:
    - Optimize game performance behind the scenes.
    - Allow developers to focus on gameplay logic rather than low-level optimizations.
7. Encourage Creativity and Experimentation:
    - Provide features that inspire unique game ideas.
    - Support unconventional mechanics, visual styles, and storytelling approaches.

By achieving these objectives, we aim to empower a wider range of developers—from beginners to experienced professionals—to create engaging and innovative games.

1.5 – JUSTIFICATION AND NEED FOR KABOOM.JS

Kaboom.js addresses these needs by providing an accessible, efficient, and community-driven platform for game development.

1. Simplicity and Accessibility:
    - **Justification**: Traditional game development frameworks can be complex, especially for beginners. Kaboom.js simplifies the process by offering an intuitive syntax and component-based system.
    - **Need**: Aspiring game developers need a friendly entry point to create games without feeling overwhelmed by technical details.
2. Rapid Prototyping and Iteration:
    - **Justification**: Game ideas often evolve during development. Kaboom.js allows quick experimentation with mechanics, visuals, and gameplay.
    - **Need**: Developers require tools that facilitate rapid prototyping and iteration to refine their game concepts efficiently.
3. Cross-Platform Compatibility:
    - **Justification**: Players use various devices—desktops, laptops, mobile phones. Kaboom.js ensures games run seamlessly across platforms.
    - **Need**: Developers seek solutions that allow them to reach a broader audience without rewriting code for each platform.
4. Efficient Asset Management:
    - **Justification**: Managing game assets (sprites, sounds, music) can be time-consuming. Kaboom.js streamlines asset handling.
    - **Need**: Developers want tools that simplify asset organization and integration, allowing them to focus on game logic.
5. Community and Collaboration:
    - **Justification**: A supportive community helps troubleshoot issues, share knowledge, and collaborate. Kaboom.js fosters this environment.
    - **Need**: Developers benefit from active communities that provide guidance, inspiration, and shared experiences.
6. Performance Optimization without Complexity:
    - **Justification**: Well-optimized games run smoothly. Kaboom.js balances performance and simplicity.
    - **Need**: Developers seek frameworks that handle performance behind the scenes, allowing them to focus on game design.

Kaboom.js simplifies game development by offering an intuitive syntax and component-based system. It enables rapid prototyping, cross-platform compatibility, and efficient asset management. The framework encourages creativity and collaboration within the game development community.

1.6 – ADVANTAGES OF KABOOM.JS

1. Intuitive Syntax and Component-Based System:
    - **Benefit**: Kaboom.js simplifies game code by offering an intuitive syntax. Developers can focus on creating game elements without getting lost in complex structures.
    - **Impact**: Beginners find it easier to grasp, and experienced developers appreciate the straightforward approach.
2. Quick Experimentation with Mechanics and Visuals:
    - **Benefit**: Kaboom.js encourages rapid prototyping. Developers can swiftly test different gameplay mechanics, visual styles, and interactions.
    - **Impact**: Iterative design becomes efficient, leading to better game concepts and faster development cycles.
3. Seamless Cross-Platform Deployment:
    - **Benefit**: Games created with Kaboom.js run consistently across platforms—whether on web browsers, desktops, or mobile devices.
    - **Impact**: Developers reach a broader audience without the hassle of platform-specific adaptations.
4. Streamlined Asset Handling:
    - **Benefit**: Kaboom.js simplifies asset management. Developers organize sprites, sounds, and music effortlessly.
    - **Impact**: Time spent on file management decreases, allowing more focus on game logic and design.

1.7 – DISADVANTAGES OF KABOOM.JS

1. Limited 3D Support:
    - **Justification**: Kaboom.js primarily focuses on 2D game development.
    - **Impact**: Developers seeking to create 3D games may need to explore other frameworks.
2. Platform Constraints:
    - **Justification**: Kaboom.js is designed for web browsers and lacks support for other platforms (e.g., mobile, console).
    - **Impact**: Developers targeting specific platforms may face limitations.
3. No Built-in Multiplayer Support:
    - **Justification**: Kaboom.js does not provide native multiplayer features.
    - **Impact**: Developers building multiplayer games must implement networking solutions externally.

1.8 – METHODOLOGY

Methodology and Challenges of Legacy Systems in Game Development

1. Understanding Legacy Systems:

- **Definition**: Legacy systems refer to outdated software and technology that organizations continue to use despite inherent problems.
- **Integration**: These systems are deeply integrated into game development pipelines, making replacement seem daunting.

2. Common Problems with Legacy Systems:

- **Security Risks**: Legacy systems are vulnerable to cyberattacks and data breaches due to outdated security measures.
- **Expensive Maintenance**: Maintaining legacy game engines consumes significant time and resources.
- **Technical Debt**: Accumulated inefficiencies and outdated code lead to technical debt.
- **Lack of Integration**: Legacy systems struggle to integrate with modern tools and technologies.
- **Falling Out of Compliance**: Compliance requirements evolve, and legacy systems may no longer meet them.

3. Impact on Game Development Strategy:

- **Exposed Weaknesses**: The gaming industry's rapid evolution highlights weaknesses in digital strategies, often tied to reliance on legacy systems.
- **Business Agility**: Legacy systems hinder agility, hindering adaptation to market changes.

Kaboom.js: A Solution for Modern Game Development

1. Lightweight and Beginner-Friendly:

- **Benefit**: Kaboom.js is fast, lightweight, and beginner-friendly.
- **Impact**: Developers can create games without high-end hardware or extensive optimization efforts.

2. Cross-Platform Compatibility:

- **Benefit**: Kaboom.js ensures seamless game deployment across web, desktop, and mobile.

- **Impact**: Developers reach wider audiences without rewriting code for each platform.

3. Streamlined Asset Handling:

- **Benefit**: Kaboom.js simplifies asset management (sprites, sounds, music).
- **Impact**: Developers focus on game logic, not file organization.

4. Active Community and Collaboration:

- **Benefit**: Kaboom.js fosters a supportive community.
- **Impact**: Developers share knowledge, troubleshoot issues, and collaborate on projects.

In summary, Kaboom.js empowers game developers by providing an accessible, efficient, and community-driven platform, addressing legacy system challenges.

CHAPTER 2: STUDY OF THE PRODUCT

2.1 – NEW PRODUCT DEVELOPMENT (NPD)

New Product Development (NPD) Report for "Hunter Mommy" Game

Introduction

"Hunter Mommy" is an exhilarating survival adventure game inspired by the beloved cartoon character Shinchan. this game promises not only entertainment but also a challenging experience that tests players' quick thinking and strategic skills.

Game Overview

- **Title:** Hunter Mommy: A Survival Adventure in the Whimsical World of Shinchan
- **Developer:** Aryan Kaul
- **Genre:** Survival
- **Platform:** Android (APK available), Web Application (For All The Devices)

Key Features

1. **One Playable Characters:**
    - **Shin Chan:** Players navigate Shinchan through emerging dangers, striving for survival.
2. **Survival Gameplay:**
    - The whimsical world is filled with obstacles.
    - Players compete to determine who survives the longest in obstacle-filled, randomly generated maps.
3. **Responsive Layout:**
    - Enjoy seamless play across various devices.
    - The game adapts to different screen sizes for a consistent experience.
4. **APK for Android:**
    - Conveniently available for all Android users.
    - Easy installation and accessibility.

Market Analysis

- **Market Size and Trends:**
    - Recently small sized games are on trend in the market, The user wants to play a quick game without any lag, Hunter Mommy is based on Web so all the processing takes place on server end which leads to a lag free

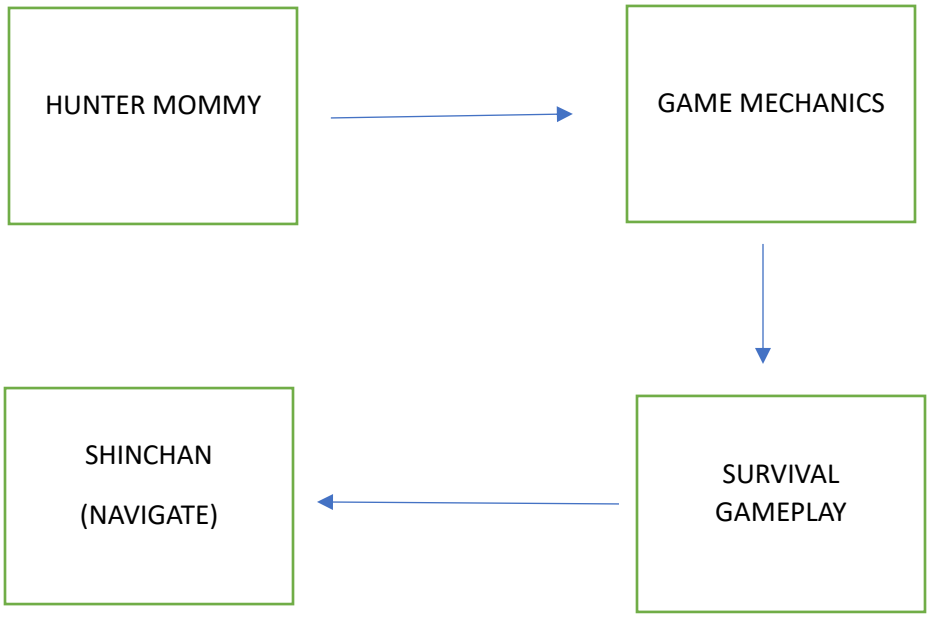
environment. It's APK is also very optimized with the size of 3-4 MB only.
    - "Hunter Mommy" fits in casual & survival games.
- **Competitor Analysis:**
    - There are many games in the market with the same genre based on the player survival but they are not as optimized as "Hunter Mommy"

Game Metrics and Performance

- **User Acquisition:**
    - User can play the game by opening the link anytime on any device.
    - Android users have a 3 MB APK for their mobiles.
    - By adding more stages and characters which were remembered by the Youth over time.
- **Monetization:**
    - The game takes 0 investment in the development process.
    - No game development engines were used, No Loyalty Fees.
    - All the revenue from ads is of the developer (100%)

Conclusion

"Hunter Mommy" holds immense potential in the gaming market. With its unique gameplay, engaging characters, and strategic challenges, it's poised to captivate players worldwide. Stay committed to continuous improvement, gather player feedback, and watch your game thrive!

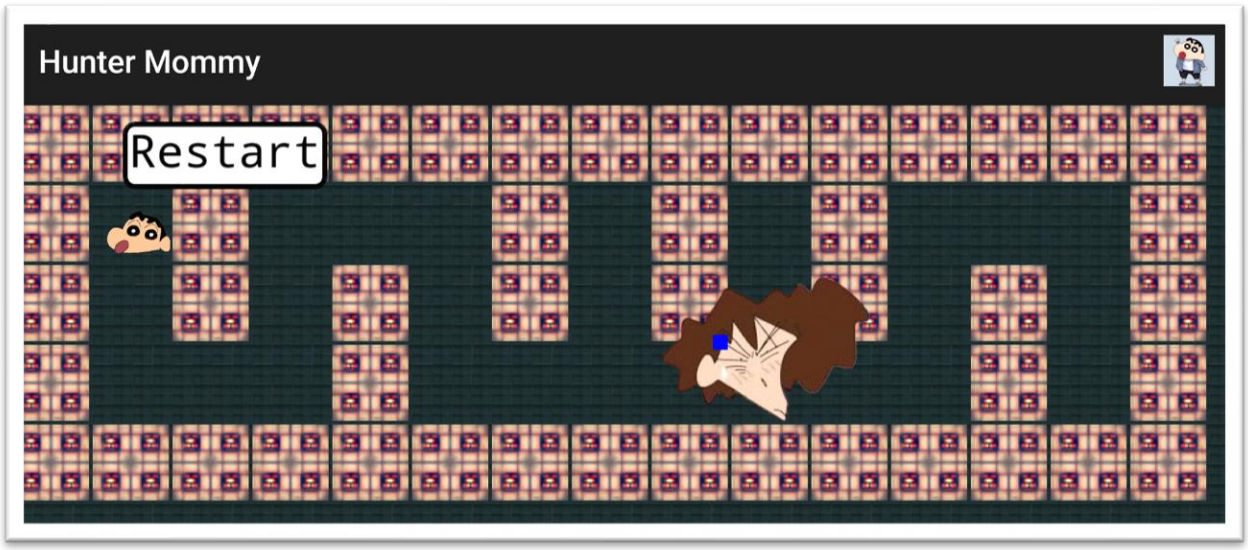


2.2 – PRODUCT LIFE CYCLE (PLC)

There are several things involved in Product Life Cycle which are as follows:

1. **Introduction and Idea Generation**:
    - **Brainstorming Ideas**:
        - Gathering a team or work individually to generate creative ideas for the game.
        - Considering the game's theme, characters, and gameplay mechanics.
    - **Unique Selling Points (USPs)**:
        - Identifying what sets "Hunter Mommy" apart from other survival games.
        - features like the whimsical Shinchan world, responsive gameplay, and the two playable characters.
    - **Inspiration from Shinchan**:
        - Explaining how the beloved cartoon character Shinchan influenced the game concept.
2. **Market Research and Analysis**:
    - **Gaming Industry Trends**:
        - Includes the research of current trends in the gaming market.
        - Understanding player preferences (e.g., quick, lightweight games).
    - **Competitor Analysis**:
        - Study existing survival games.
        - Identifying their strengths and weaknesses.
        - Position "Hunter Mommy" as optimized and unique.
3. **Game Design and Mechanics**:
    - **Core Gameplay Rules**:
        - Save the Shinchan from the hunter – Shinchan's Mom
    - **Survival Mechanics**:
        - obstacles and challenges enhance the player experience by adding difficulties to the game.
        - the randomly generated maps – For loop iteration.
    - **Playable Characters**:
        - Introducing Shin Chan and Shinchan's Mom.
        - Shinchan has the limitation that he cant bypass the obstacles but on the other hand hunter is free to move.
4. **Development and Testing**:
    - **Kaboom.js Development**:
        - the development process using Kaboom.js.
        - Installing Assets & scaling them accordingly
    - **Thorough Testing**:

        - Testing the game across different devices and operating system. For ensuring smooth gameplay.
        - Balancing gameplay elements based on the game experience (difficulty, pacing).
    - **Device Optimization**:
        - the game adapts to different devices (web, mobile).
5. **Post-Launch and Iteration**:
    - **Player Feedback**:
        - player feedback is collected (reviews, surveys).
        - Address any reported issues promptly.
    - **Regular Updates**:
        - Planning updates to keep the game fresh.
        - Adding new content, fix bugs, and enhance gameplay.
6. **Maturity and Sustained Growth**:
    - **Metrics Monitoring**:
        - Track user retention, revenue, and engagement.
        - Adjusting strategies based on data.
    - **Expansion Opportunities**:
        - Considering adding new levels, features, or platforms time to time.
7. **Decline and Sunsetting**:
    - **Graceful Sunset**:
        - Acknowledging that all games eventually decline.
        - Deciding whether to continue supporting the game or explore new projects.



2.3 – PRODUCT DESIGN

Game Overview

Game Name and Logo:

- The game will be titled **"Hunter Mommy: Shinchan's Survival Adventure."**
- The logo will feature vibrant illustrations of Shinchan and his mom in their game characters, capturing the whimsical essence of the game.

Game Interface

The game interface will be designed to be both user-friendly and visually appealing. Here are the key elements:

1. **Start/Restart Button**:
    - Initiates or restarts the game.
    - Provides a seamless entry point for players.
2. **Pause/Settings Button**:
    - Allows players to pause the game or adjust settings.
    - Ensures flexibility during gameplay.
3. **Scoreboard**:
    - Displays the current scores for both characters (Shinchan and his mom).
    - Keeps players informed about their progress.
4. **Touch Control**:
    - Enables precise control over Shinchan's movements.
    - Ensures smooth navigation through the game world.
5. **Action Buttons for Shinchan**:
    - Buttons for actions such as jumping, crawling, or using items.
    - Enhances gameplay dynamics.
6. **Gun Control for Shinchan's Mom**:
    - Allows Shinchan's mom to aim and shoot.
    - Adds an exciting element to the game.

Game Maps

- The game maps will be **randomly generated** to provide a fresh experience each time.
- While they will be obstacle-free, players will encounter various threats, such as wild animals or dangerous traps.
- Exploration and adaptability will be key to survival.

Characters

- The characters will be designed in a **cartoonish style**, paying homage to the original Shinchan series.
- **Shinchan**:
  - Agile and quick.
  - Navigates obstacles swiftly.
- **Shinchan's Mom**:
  - Strong and accurate with her gun.

Sound and Music

- The game will feature:
  - **Upbeat Music**: Sets the tone for the adventure.
  - **Sound Effects**: Includes gunshots, animal noises, and background sounds.
  - Enhances immersion and engagement.

Game Updates

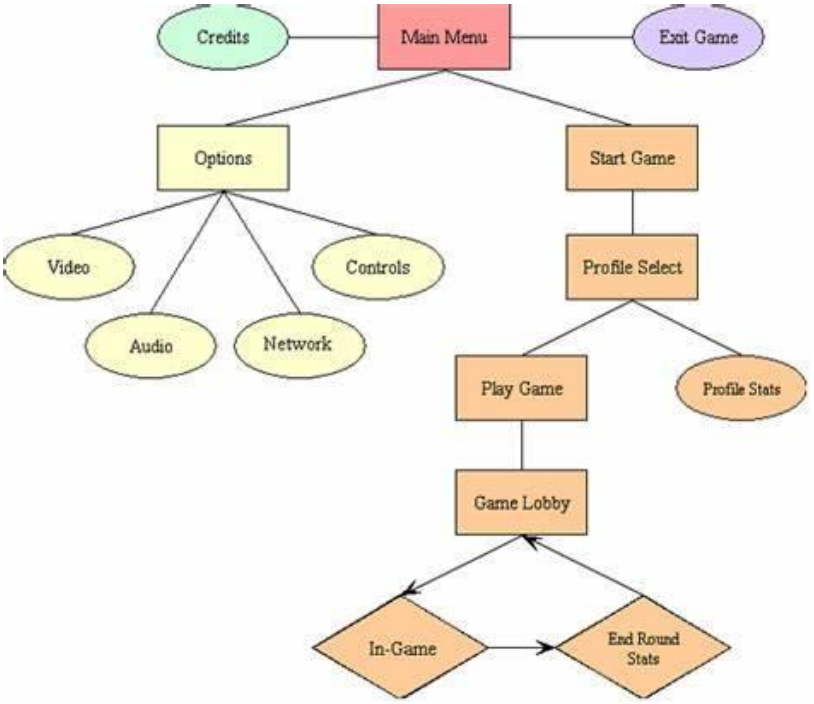
- Regular updates will be rolled out to:
  - **Fix Bugs**: Ensure a smooth gaming experience.
  - **Add New Features**: Keep players engaged.
  - **Introduce Challenges**: Maintain excitement.

Accessibility

- The game will be designed to be accessible to players of all ages and abilities:
  - **Easy-to-Use Controls**: Intuitive controls for a seamless experience.
  - **Adjustable Difficulty Levels**: Accommodates both casual and hardcore players.

APK for Android

- The game will be available as an **APK for Android devices**:
  - Lightweight and optimized for mobile play.
  - Ensures consistent performance across devices.



2.4 – OPTIMIZATION IN DESIGN

To enhance the performance and overall player experience of "Hunter Mommy," below mentioned optimization strategies should be used:

1. **Optimize Graphics**:
    - Use efficient graphic design techniques to reduce the number of polygons and textures used in the game.
    - Consider the following:
        - **Texture Compression**: Compress textures to reduce memory usage without compromising visual quality.
        - **Sprite Atlases**: Combine multiple small textures into a single atlas to minimize draw calls.
        - **Dynamic Loading**: Load textures and assets only when needed to reduce initial load time.
2. **Reduce Draw Calls**:
    - Each draw call involves switching shaders, textures, and objects, which can impact performance.
    - Optimize by:
        - **Batching**: Combine similar objects into a single draw call.

        - **Instancing**: Use GPU instancing for repeated objects with the same material.
3. **Implement Level of Detail (LOD)**:
    - Vary the level of detail based on the distance from the camera.
    - Simplify models or use lower-resolution textures for distant objects.
    - Improve performance, especially on lower-end devices.
4. **Optimize Physics**:
    - Physics calculations can be CPU-intensive.
    - Consider:
        - **Bounding Volumes**: Use simple shapes (e.g., spheres, boxes) for collision detection.
        - **Asynchronous Physics**: Offload physics calculations to separate threads.
5. **Use Caching**:
    - Cache frequently used data and assets in memory:
        - **Texture Streaming**: Load textures progressively during gameplay.
        - **Object Pooling**: Reuse objects (e.g., bullets, enemies) instead of creating new ones.
6. **Optimize Audio**:
    - Audio processing can impact performance:
        - **Audio Compression**: Use compressed audio formats (e.g., OGG) to reduce memory usage.
        - **Spatial Audio**: Limit the number of simultaneous audio sources.
        - **Streaming**: Load audio clips on demand.
7. **Memory Usage Optimization**:
    - Efficient data structures and algorithms reduce memory overhead:
        - **Garbage Collection**: Minimize memory allocations to avoid frequent garbage collection.
        - **Memory Pools**: Pre-allocate memory for frequently used objects.
8. **Network Traffic Optimization** (for multiplayer features):
    - Reduce data transfer between devices:
        - **Delta Compression**: Send only changed data during updates.
        - **Client-Side Prediction**: Predict client actions to reduce server communication.
9. **AI Optimization**:
    - AI calculations can impact CPU performance:
        - **Pathfinding**: Optimize pathfinding algorithms (e.g., A*).
        - **Decision Trees**: Simplify decision-making processes.
        - **Behavior Trees**: Efficiently manage AI behaviors.

2.5 – CHALLENGES FACED

Certainly! Let's format the challenges faced during the game development process:

1. **Responsive Design Across Devices**:
    - Ensuring that the game works seamlessly on various screen sizes (mobile, tablet, desktop).
    - Adapting UI elements, fonts, and layouts dynamically.
2. **Rendering Optimization**:
    - Balancing visual quality with performance.
    - Reducing draw calls, texture sizes, and shader complexity.
    - Implementing efficient rendering techniques.
3. **Pixel Calculations for Asset Adjustment**:
    - Precisely aligning assets (sprites, UI elements) to pixel grids.
    - Handling different resolutions and aspect ratios.
    - Avoiding subpixel rendering artifacts.
4. **Custom Asset Design**:
    - Creating unique and visually appealing assets:
        - **Characters**: Designing Shinchan, his mom, and other game elements.
        - **Backgrounds**: Crafting whimsical environments.
        - **Icons and Buttons**: Custom UI elements.
5. **Physics Integration**:
    - Adding physics interactions to the game world:
        - **Collisions**: Ensuring characters interact realistically with walls and obstacles.
        - **Gravity**: Implementing gravity for jumps and falls.
        - **Rigid Bodies**: Handling dynamic objects (e.g., falling rocks).

CHAPTER 3: FEASIBILITY STUDY

3.1 – TECHNICAL FEASIBILITY

Technical Feasibility Study for "Hunter Mommy" Game:

1. Overview:

"Hunter Mommy" is an action-adventure game where players take on the role of a fierce mom who hunts down mythical creatures to protect her family. The game will feature sprite-based graphics, physics-based movement, and collision detection.

2. Kaboom.js Features:

- **Ease of Use**: Kaboom.js provides a straightforward API for creating game objects, handling collisions, and managing game states.
- **Sprite Rendering**: Kaboom.js allows rendering sprites, which is essential for character animations, enemies, and environment elements.
- **Physics Engine**: The built-in physics engine handles gravity, collisions, and movement.
- **Event Handling**: Kaboom.js supports event listeners for keyboard input, mouse interactions, and game updates.
- **Component-Based Architecture**: Assemble game objects from reusable components (e.g., sprite, position, area, body) to keep code organized and modular[1].

3. Technical Considerations:

- **Game Loop**: Kaboom.js manages the game loop, including update and draw phases. Developers can register functions to run during these phases.
- **Collision Detection**: Utilize the `area()` component for collision detection between characters, enemies, and environment objects.
- **Physics Simulation**: The `body()` component provides physics simulation (e.g., gravity, velocity, acceleration).
- **Sprites and Animations**: Load sprite sheets or individual images for characters, enemies, and items. Animate sprites using frame sequences.
- **Game States**: Implement game states (e.g., menu, gameplay, game over) using Kaboom.js functions like `onUpdate()` and `onDraw()`.
- **Performance Optimization**: Optimize sprite loading, minimize unnecessary calculations, and manage memory efficiently.

3.2 – OPERATIONAL FEASIBILITY

Operational feasibility refers to the practicality of implementing and maintaining a system or project. Let's explore the operational feasibility of creating the block-breaking game using **Kaboom.js**:

1. **Technical Resources:**
    - **Hardware**: Developing a game with Kaboom.js doesn't require high-end hardware. A standard computer with a web browser and code editor is sufficient.
    - **Software**: Kaboom.js is a JavaScript library, so you'll need a code editor (e.g., Visual Studio Code, Replit) and a web browser (Chrome, Firefox, etc.).
2. **Development Effort:**
    - **Ease of Learning**: Kaboom.js is beginner-friendly, especially for those familiar with JavaScript. Its simple syntax and component-based system reduce the learning curve.
    - **Development Time**: Creating a basic block-breaking game can be done relatively quickly. However, adding features (power-ups, levels) will extend the development time.
3. **Maintenance and Updates:**
    - Kaboom.js has an active community and regular updates. Staying informed about new features and bug fixes is essential.
    - As your game evolves, maintenance tasks (bug fixes, performance optimization) will be necessary.
4. **Testing and Debugging:**
    - Kaboom.js provides tools for testing and debugging (e.g., collision detection, console logs).
    - Regular testing during development ensures a stable game.
5. **Deployment and Hosting:**
    - Deploying a Kaboom.js game is straightforward. You can host it on platforms like GitHub Pages, Netlify, or Replit.
    - Ensure compatibility across different browsers.
6. **User Experience:**
    - Consider user feedback and playtesting. Is the game enjoyable? Are there any usability issues?
    - Optimize performance to ensure smooth gameplay.
7. **Legal and Licensing:**
    - Ensure compliance with Kaboom.js's licensing terms.
    - If using external assets (sprites, sounds), verify their licenses and give proper credit.

3.3 – BEHAVIORAL FEASIBILITY

The Behavioral Feasibility of creating a game using **Kaboom.js**. This aspect focuses on how well the game aligns with user expectations, engages players, and provides an enjoyable experience:

1. **User Engagement:**
    - **Positive Experience**: Kaboom.js simplifies game development, allowing developers to focus on creativity rather than technical complexities. Players appreciate games that are easy to pick up and play.
    - **Intuitive Controls**: Ensure that game controls (e.g., keyboard, mouse) are straightforward and responsive. Players should feel in control.
2. **Game Mechanics:**
    - **Gameplay Flow**: The block-breaking game should have a smooth flow. Introduce players to the mechanics gradually (e.g., start with a single ball and paddle).
    - **Challenging Yet Fair**: Balance difficulty levels. Too easy, and players lose interest; too hard, and they become frustrated.
3. **Visuals and Audio:**
    - **Graphics**: Use visually appealing sprites and animations. Consistent art style enhances immersion.
    - **Sound Effects and Music**: Well-chosen sound effects (e.g., ball hitting blocks, paddle movement) contribute to the overall experience.
4. **Feedback and Rewards:**
    - **Feedback Loop**: Provide immediate feedback (e.g., visual effects, sounds) when the player performs actions (hits a block, loses a life).
    - **Rewards**: Celebrate achievements (e.g., breaking all blocks, reaching a new level) to motivate players.
5. **Progression and Replayability:**
    - **Levels**: Design multiple levels with increasing difficulty. Each level should introduce new challenges or features.
    - **High Score**: Include a scoring system and encourage players to beat their own or others' high scores.
6. **Social and Multiplayer Aspects:**
    - **Leaderboards**: Consider integrating online leaderboards to foster competition.
    - **Multiplayer**: If feasible, add local multiplayer (e.g., two-player mode).
7. **Testing and Iteration:**
    - **Playtesting**: Involve others (friends, family, or fellow developers) to playtest the game. Their feedback is invaluable.
    - **Iterate**: Be open to making improvements based on user feedback.

3.4 – ECONOMIC FEASIBILITY

1. **Development Costs:**
    - **Software and Tools**: Kaboom.js is open-source and free to use. You'll need a code editor (e.g., Visual Studio Code, Replit) and a web browser for testing.
    - **Asset Creation**: Consider costs related to creating or acquiring game assets (sprites, sounds, music). You can find free or paid assets online.
2. **Monetization Strategies:**
    - **Free-to-Play (F2P)**: Release the game for free and monetize through ads, in-game purchases (e.g., power-ups, skins), or donations.
    - **Premium Model**: Charge an upfront fee for the game. Ensure the game's quality justifies the price.
    - **Freemium**: Offer a basic version for free and provide additional content or features as in-app purchases.
    - **Subscriptions**: Consider subscription-based models for ongoing revenue.
3. **Market Research:**
    - Analyze the market for block-breaking games. Is there demand? Who are your potential players?
    - Study competitors and identify unique selling points for your game.
4. **Revenue Projections:**
    - Estimate potential revenue based on the chosen monetization model.
    - Consider user acquisition costs (if advertising) and retention rates.
5. **Operational Costs:**
    - **Hosting**: If you host the game online, factor in hosting costs (e.g., GitHub Pages, Netlify).
    - **Maintenance**: Regular updates, bug fixes, and improvements require time and effort.
6. **ROI (Return on Investment)**:
    - Calculate the return on investment by comparing development costs to projected revenue.
    - Consider both short-term and long-term returns.
7. **Marketing and Promotion:**
    - Allocate resources for marketing efforts (social media, game forums, influencers).
    - Build a community around your game.
8. **Legal and Licensing:**
    - Understand licensing requirements for Kaboom.js and any external assets used.
    - Protect your intellectual property (e.g., copyright, trademarks).

9. **Sustainability and Longevity:**
    - Ensure the game's appeal lasts beyond the initial launch.
    - Plan for updates, seasonal events, or expansions.
10. **Risk Assessment:**
    - Identify risks (e.g., changing market trends, technical challenges) and develop mitigation strategies.

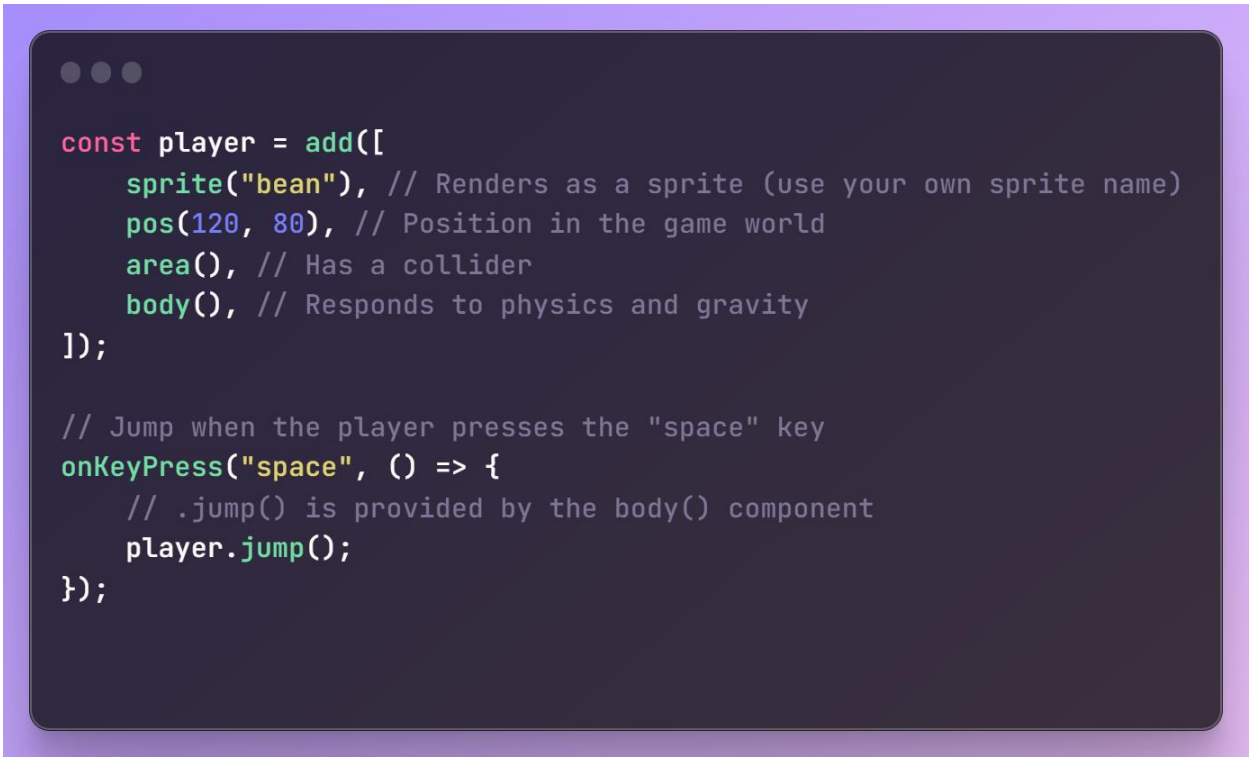
CHAPTER 4: SOFTWARE REQUIREMENT & SPECIFICATION

4.1 – SOFTWARE REQUIREMENT

Kaboom.js is a versatile JavaScript game programming library that makes game development fast and enjoyable. Here are the software requirements and steps to get started:

1. **Web Browser**:
   - Kaboom.js can be used with any modern web browser.
   - No additional software or libraries are required.
2. **Code Editor (Optional)**:
   - While not mandatory, having a code editor (e.g., Visual Studio Code, Replit) can enhance your development experience.
   - You can write Kaboom.js code directly in the browser if you prefer.
3. **Kaboom.js Library**:
   - You can include the Kaboom.js library in your project using one of the following methods:
     - **CDN Provider (Recommended)**:
       - The quickest way is to get the package from a CDN provider (e.g., unpkg, jsdelivr).
       - Include the following script tag in your HTML file:

     - **NPM (Node Package Manager)**:
       - Install Kaboom.js using npm:
       - `npm install kaboom`
       - You can then import it into your project.
4. **Initializing Kaboom Context**:
   - To start using Kaboom.js, initialize the context by calling `kaboom()`:
5. **Creating Your First Game Object**:
   - Define game objects by assembling components.
   - For example, to create a player character with a sprite, position, collider, and physics:



```
const player = add([
    sprite("bean"), // Renders as a sprite (use your own sprite name)
    pos(120, 80), // Position in the game world
    area(), // Has a collider
    body(), // Responds to physics and gravity
]);

// Jump when the player presses the "space" key
onKeyPress("space", () => {
    // .jump() is provided by the body() component
    player.jump();
});
```

6. **Customize Your Game**:
    - Explore other components (e.g., health, lifespan, events) to add functionality to your game objects.

4.2– SOFTWARE SPECIFICATIONS

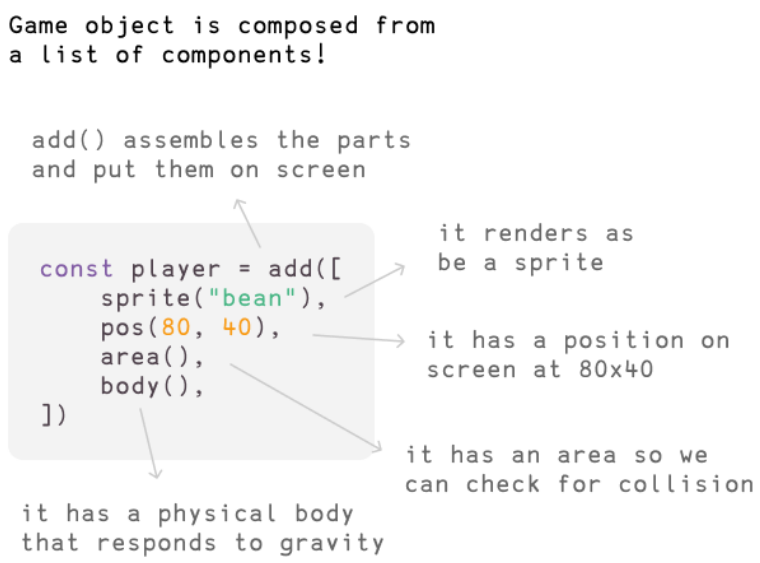
Let's dive into the software specifications for creating games using **Kaboom.js**, a powerful JavaScript game programming library. Below, I'll outline key components, functions, and concepts you'll encounter when working with Kaboom.js:

1. **Initialization and Context Setup**:
    - To start using Kaboom.js, initialize the context using `kaboom()`. This sets up the game environment and provides access to various functions and components.
    - Example:
    - `kaboom();`
2. **Components**:
    - Components are building blocks for game objects. You assemble game objects by combining components.
    - Common components include:

- `sprite()`: Renders an image or sprite.
- `pos()`: Defines the position of the game object.
- `area()`: Adds collision detection.
- `body()`: Handles physics and gravity.
- `health()`: Manages health points.
- `lifespan()`: Controls how long an object exists.

3. **Creating Game Objects**:
   - Use the `add()` function to create game objects from components.
   - Example:

```
const player = add([
  sprite("bean"), // Sprite name
  pos(120, 80), // Position
  area(), // Collider
  body(), // Physics
]);
```



4. **Event Handling**:
   - Register events for game objects using functions like `onUpdate()`, `onKeyPress()`, and `onCollide()`.

- Example:
- ```
  onKeyPress("space", () => {
      player.jump(); // Jump when space key is pressed
  });
  ```

5. **Customization and Behavior**:
   - Customize game objects by adjusting properties (e.g., speed, color, opacity).
   - Define behaviors using components and event handlers.
6. **Global Functions**:
   - Functions like `quit()`, `vec2()`, and `canvas` are available globally after calling `kaboom()`.
   - You can also prevent global imports by using a context handle (e.g., `const k = kaboom({ global: false })`).
7. **Game Loop**:
   - Kaboom.js manages the game loop automatically. Your `onUpdate()` functions run every frame.
8. **Assets and Resources**:
   - Load sprites, sounds, and fonts using functions like `loadSprite()`, `loadSound()`, and `loadFont()`.
9. **Scenes and Levels**:
   - Organize your game into scenes (e.g., main menu, gameplay, game over).
   - Transition between scenes using `go()`, `addLevel()`, and `destroyAll()`.
10. **Debugging and Testing**:
    - Use the browser console for debugging.
    - Test collision behavior, physics, and interactions.

4.3 – TECHNOLOGY SPECIFIC REQUIREMENT

Technology-Specific Requirements for Kaboom.js

1. On Local Machine

When working with Kaboom.js on your local machine, consider the following requirements:

a. Basic Understanding of JavaScript

Before you start building games with Kaboom.js, ensure that you have a foundational understanding of JavaScript. Familiarize yourself with concepts such as variables,
  ```

functions, loops, and conditional statements. This knowledge will be essential as you write game logic and interact with the Kaboom.js library.

b. Web Browser and HTML Document

1. **Web Browser**: You'll need a web browser (such as Chrome, Firefox, or Edge) to run your Kaboom.js games. Make sure your browser is up-to-date to take advantage of modern features and performance improvements.
2. **HTML Document**: Create an HTML file where you'll write your game code. You can either create a new HTML file specifically for your game or use an existing one. Include the necessary script tags to load Kaboom.js.

2. On Virtual Machines

If you're working with Kaboom.js on virtual machines (VMs), consider the following:

a. Setting Up the VM Environment

1. **Choose a VM Solution**: Select a virtualization platform such as VirtualBox, VMware, or Hyper-V. Install the chosen VM software on your host machine.
2. **Create a VM**: Set up a new virtual machine with the desired operating system (e.g., Ubuntu, Windows, or macOS). Allocate sufficient resources (CPU, RAM, and storage) based on your game development needs.
3. **Install Node.js**: Kaboom.js requires Node.js to run. Install Node.js on your VM by following the official installation instructions for your OS.
4. **Install a Text Editor**: Choose a text editor (e.g., Visual Studio Code, Sublime Text, or Atom) to write your game code within the VM.

b. Accessing the VM

1. **SSH or Remote Desktop**: Access your VM using SSH (for Linux-based VMs) or Remote Desktop (for Windows-based VMs). Ensure that you can connect to the VM from your host machine.
2. **Shared Folders**: Set up shared folders between your host machine and the VM. This allows you to easily transfer files (including your game code) back and forth.

3. Additional Resources

- Explore the official Kaboom.js documentation for comprehensive guides, examples, and best practices.

CHAPTER 5: TECHNOLOGY & DEVELOPMENT ENVIRONMENT

5.1 – ABOUT TECHNOLOGY

Kaboom.js is a JavaScript game programming library that enables developers to create games quickly and enjoyably. Whether you're an experienced developer or a beginner, Kaboom.js simplifies game development by providing an intuitive API. With Kaboom.js, you assemble game objects from a list of components, each offering specific functionalities like rendering, physics, and collision detection. It emphasizes creativity, simplicity, and fun, making it accessible for developers of all levels.

Here are the key points about Kaboom.js:

1. **Fast and Fun Development**:
    - Kaboom.js simplifies game development by offering an intuitive API. It allows you to focus on creativity rather than boilerplate code.
    - With Kaboom.js, you can assemble game objects from a list of components, each providing specific functionalities like rendering, physics, collision detection, and more.
2. **Basic Concepts**:
    - Assemble game objects using components: For example, create a player character with a sprite, position, collider, and physics behavior.
    - Respond to events: Register functions to run on specific events (e.g., frame updates, key presses, collisions).
    - Set up gravity, load sprites, and define game behavior.
3. **Example Usage**:
    - Initialize Kaboom context: Start your game with `kaboom()`.
    - Define gravity: Use `setGravity(2400)` to add realistic physics.
    - Load default sprites: For instance, `loadBean()` loads a default sprite.
    - Create game objects: Assemble components (e.g., `sprite`, `pos`, `area`, `body`) to build characters, obstacles, and more.
    - Handle events: Respond to key presses (e.g., jump when the player presses the "space" key).

In conclusion, **Kaboom.js** is a fantastic choice for anyone looking to dive into game development. Its intuitive API, emphasis on creativity, and efficient component-based approach make it accessible to both seasoned developers and beginners. Whether you're building a platformer, shooter, or puzzle game, Kaboom.js provides the tools you need to bring your game ideas to life.

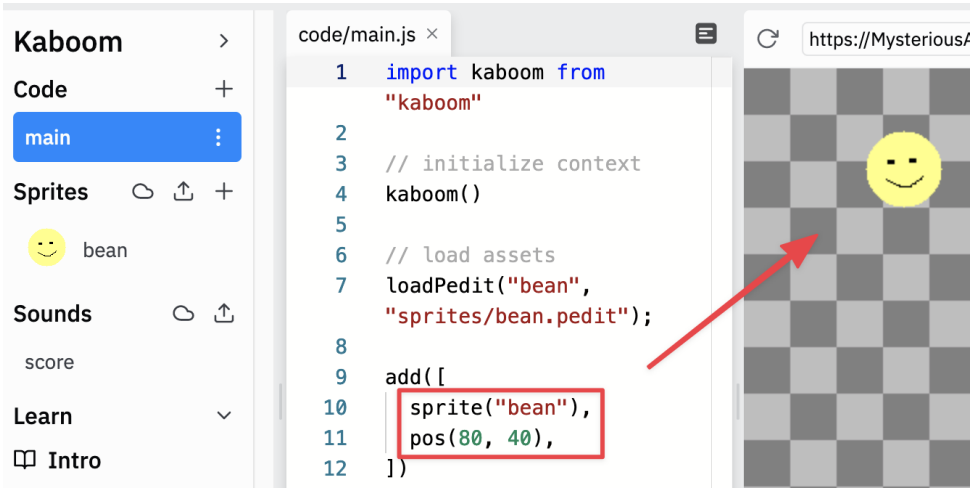
5.2 – ABOUT DEVELOPMENT ENVIRONMENT

Replit's appeal lies in its seamless and accessible web-based IDE that caters to the needs of a wide audience, from coding novices to seasoned professionals. By eliminating the complexities of local installations and intricate setups, Replit streamlines the development process, providing a user-friendly interface that resonates with learners, educators, and industry experts alike. Its compatibility with various programming languages and frameworks ensures versatility, accommodating diverse project requirements.

A standout feature is Replit's integration with Kaboom.js, a library renowned for simplifying game development. Kaboom.js empowers developers to effortlessly piece together game elements, manage events, and craft immersive gaming experiences. With this integration, Replit becomes an invaluable space for experimenting with game development, enabling users to seamlessly write, run, and share game code. This inclusivity caters to both newcomers seeking an introduction to game development and experienced developers looking for an efficient environment.

The collaborative capabilities of Replit enhance its utility further. By enabling multiple users to work simultaneously on the same project, the platform fosters a collaborative atmosphere that encourages teamwork and knowledge exchange. This dynamic collaboration is particularly advantageous for educators conducting real-time coding exercises and facilitating engaging projects for students.

In summary, Replit not only serves as a comprehensive online coding and testing environment but also extends its functionality to support game development through Kaboom.js. Its simplicity and accessibility make it an invaluable tool for a diverse user base, bridging the gap between learners, educators, and professionals in the ever-evolving field of software development.



5.3 – LANGUAGES USED

5.3.1

HTML (HyperText Markup Language): Building Blocks of the Web

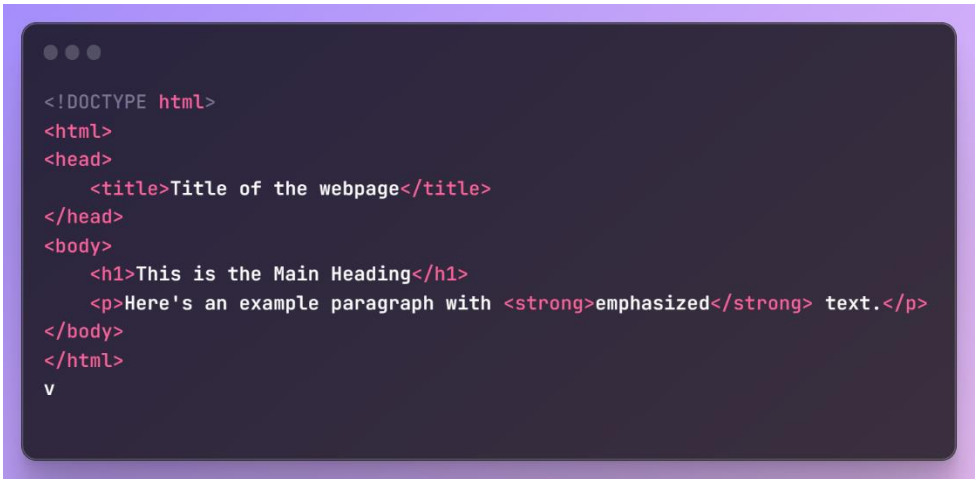
HTML (HyperText Markup Language) is the standard markup language for documents designed to be displayed in a web browser. It defines the content and structure of web content. When you visit a website, your web browser receives HTML documents from a web server or local storage and renders them into multimedia web pages. Here are some key points about HTML:

- **Elements and Tags**: HTML elements are the building blocks of web pages. They are delineated by tags written using angle brackets. For example, `<img>` introduces an image, and `<p>` surrounds a paragraph of text. Browsers interpret these tags to display content.

- **Semantics and Structure**: HTML provides a means to create structured documents by denoting semantic elements such as headings, paragraphs, lists, links, and quotes. It originally included cues for appearance, but modern best practices separate styling using Cascading Style Sheets (CSS).

- **Interactivity**: HTML can embed programs written in scripting languages like JavaScript, affecting the behavior and content of web pages. It allows developers to create dynamic and interactive experiences.

- **HTML5**: HTML5, the latest version, extends HTML to support video, audio, and canvas-based graphics. It's used alongside JavaScript for modern web applications.



```
<!DOCTYPE html>
<html>
<head>
    <title>Title of the webpage</title>
</head>
<body>
    <h1>This is the Main Heading</h1>
    <p>Here's an example paragraph with <strong>emphasized</strong> text.</p>
</body>
</html>
v
```

5.3.2 – JAVASCRIPT

JavaScript, often hailed as a lightweight, cross-platform, single-threaded, and interpreted compiled programming language, extends its influence beyond its role as the scripting language for webpages. While renowned for enhancing interactivity on the client side, JavaScript's capabilities span both client-side and server-side realms.

On the client side, JavaScript empowers developers to manipulate HTML elements and respond to user events, thereby elevating webpage interactivity. Notable client-side libraries such as AngularJS, ReactJS, and VueJS leverage JavaScript's capabilities. On the server side, JavaScript finds a robust presence through frameworks like Node.js, enabling seamless server-side development, communication with databases, and file manipulations.

JavaScript navigates the dichotomy between imperative and declarative programming paradigms. In its imperative role, JavaScript controls the flow of computation, offering procedural and object-oriented programming approaches. Meanwhile, its declarative side shines through features like arrow functions, allowing developers to focus on describing desired results without specifying step-by-step procedures.

Linking JavaScript to HTML provides the bridge between logic and presentation. Developers can embed JavaScript directly within HTML using the `<script>` tag or opt for an external approach by placing JavaScript code in a separate `.js` file. This modular approach facilitates cleaner code organization and maintenance.

A brief glance at JavaScript's history reveals its origins in 1995 at Netscape, crafted by Brendan Eich. Originally named LiveScript, it later evolved into JavaScript. Unlike many programming languages, JavaScript operates without a formal concept of input or output, adding to its unique nature.

```
// JavaScript code to prompt user for their name and display a greeting
// Prompt the user for their name
let userName = prompt("What is your name?");

// Check if the user provided a name
if (userName) {
  // Display a personalized greeting in the console
  console.log(`Hello, ${userName}! Welcome to the world of JavaScript.`);
} else {
  // If the user did not provide a name, display a generic greeting
  console.log("Hello there! Welcome to the world of JavaScript.");
}
```

5.3.2 – TYPESCRIPT

TypeScript extends JavaScript by adding static type checking and other features. It allows developers to write more robust code while still transpiring to plain JavaScript.

1. **Static Typing**:
    - TypeScript introduces **static type checking**. This means that you can explicitly declare the types of variables, function parameters, and return values.
    - For example, you can define a variable as a string, number, boolean, or any custom type. TypeScript ensures that you don't accidentally assign incompatible values.
2. **Type Inference**:
    - TypeScript infers types based on context. If you don't explicitly specify a type, it tries to deduce it from your code.
    - For instance, if you assign a string to a variable, TypeScript automatically considers that variable as having a string type.
3. **Interfaces and Classes**:
    - TypeScript supports **interfaces** for defining contracts between different parts of your code. You can describe the shape of objects using interfaces.
    - You can also create **classes** with properties, methods, and inheritance. TypeScript enforces class structures and access modifiers.
4. **Compile Step**:
    - Unlike JavaScript, which runs directly in browsers or Node.js, TypeScript requires a **compilation step**.
    - You write TypeScript code, then use the TypeScript compiler (`tsc`) to generate JavaScript files that browsers or Node.js can execute.
5. **Tooling and IDE Support**:
    - TypeScript integrates well with popular code editors like Visual Studio Code, providing autocompletion, type hints, and error checking.
    - The TypeScript language service helps catch potential issues during development.
6. **Compatibility with Existing JavaScript**:
    - You can gradually migrate existing JavaScript projects to TypeScript. Simply rename `.js` files to `.ts`, and TypeScript will infer types where possible.
    - TypeScript allows you to leverage existing JavaScript libraries without major modifications.

5.4 – TOOLS USED

Below mentioned are some tools and websites used in the development of the game:

- VS Code
- Replit
- Itch.io – Assets
- Photoshop
- Fl Studio
- Audio Amplifier Plugins For Audio Generation
- Amazon Web Services
- Docker
- Linux

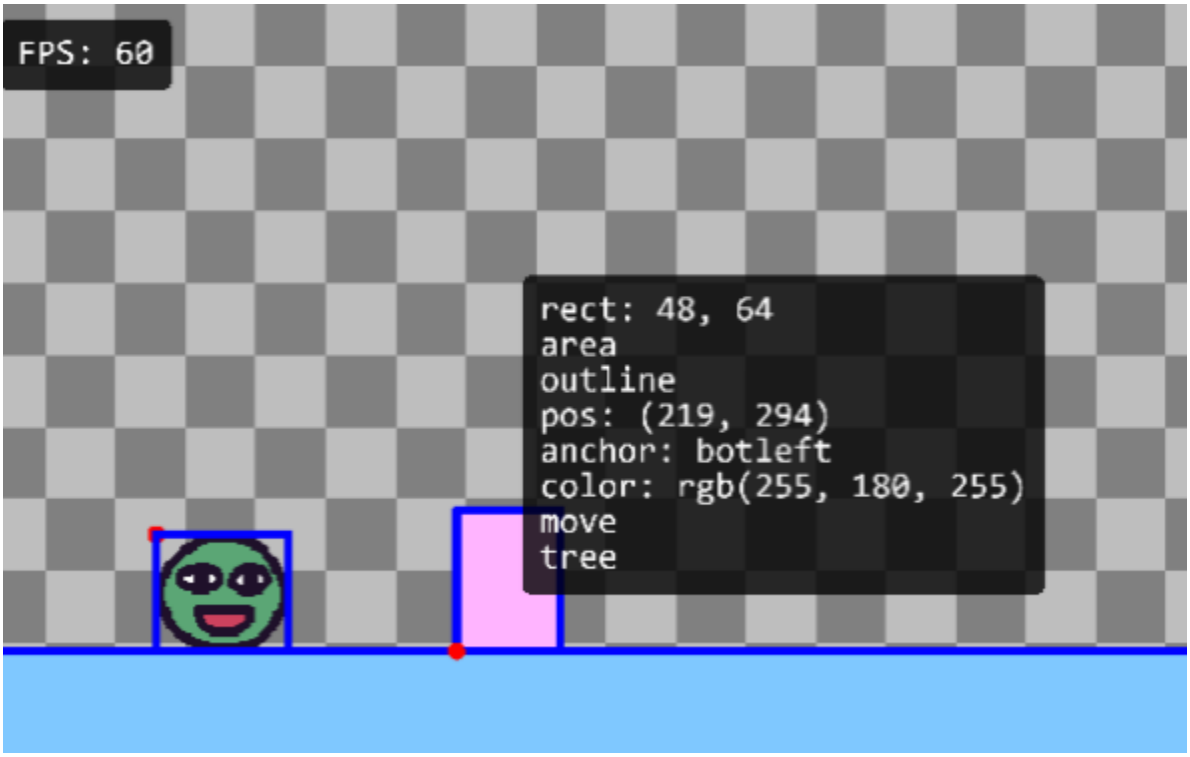
CHAPTER 6: MORPHOLOGY OF DESIGN

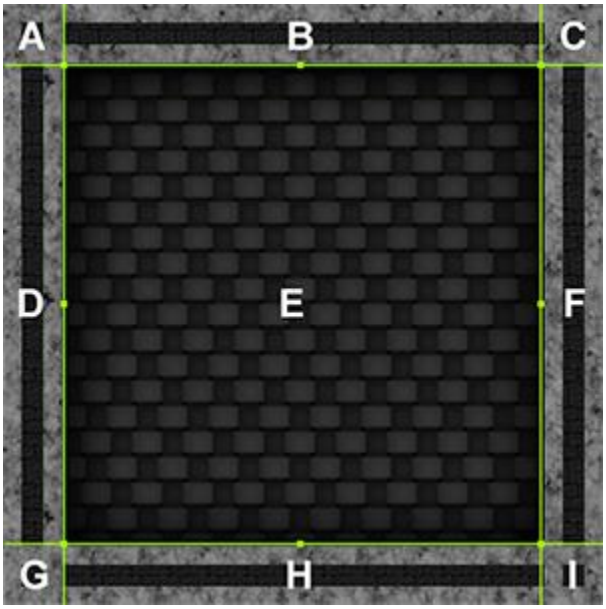
6.1 – MORPHOLOGY

Morphology of design refers to the study of the chronological structure of design projects. It involves examining the engineer's actions as they identify and solve problems throughout the design process. Let's break down the key phases within this morphology:

1. **Needs Analysis**:
    - The creation process begins by recognizing a need. This need can arise from observation, detailed study, or specific circumstances.
    - Engineers evaluate whether the need exists, whether it's realistic, and whether it's new or already present.
    - Needs analysis sets the foundation for subsequent design steps.
2. **Feasibility Study**:
    - At this stage, the product takes abstract forms. Engineers assess whether satisfying the original need is feasible.
    - Alternative solutions undergo physical and economic analyses. The goal is to establish a design concept that can be realized and accepted.
    - Examples include specifying heating, ventilation, and air conditioning requirements for a building or alternative energy supplies due to low fossil fuel reserves.
3. **Preliminary Design**:
    - The focus here is on selecting the best solution from a range of alternatives.
    - Engineers compare designs against given criteria and constraints. An open mind is essential during this phase.
    - The goal is to narrow down options and identify the most promising design.
4. **Detailed Design**:
    - This stage aims to produce a complete set of working drawings that manufacturers can use.
    - The design becomes less flexible, reflecting planning for both manufacturing and consumption stages.
    - Construction and testing of components may be necessary, and prototypes are evaluated.
5. **Production**:
    - The actual construction of the device or system takes place.
    - Planning for construction should have been incorporated into the design.
    - Knowledge of machine capabilities is crucial, and quality control measures are established.
6. **Distribution**:
    - Engineers anticipate transportation of the manufactured article, whether complete or in subassembly form.
    - Considerations include packaging, availability of vehicles, regulations, shelf life, storage facilities, and environmental conditions.

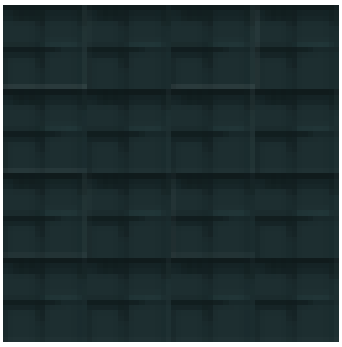
6.2 – DIAGRAMS


FPS: 60
rect: 48, 64
area
outline
pos: (219, 294)
anchor: botleft
color: rgb(255, 180, 255)
move
tree


A
B
C
D
E
F
G
H
I

6.3 – ELEMENTS USED

BG TILES –



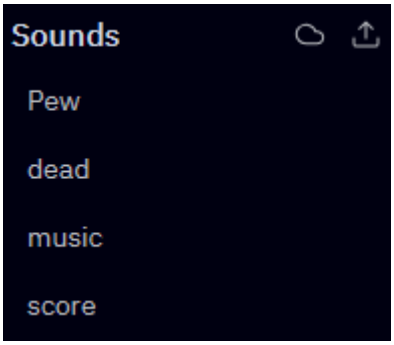
HUNTING CHARACTER (SHINCHAN'S MOM) -

WALLS –

PLAYING CHARACTER (SHINCHAN) –

MENU BACKGROUND –

SOUNDS LIST -



CHAPTER 7: CODING OF THE PROJECT

7.1 – STRUCTURE OF THE PROJECT DIRECTORY

In light of the gave setting, the venture catalog of this Replit Kaboom.js layout has the accompanying design:

1. **'code' Folder**:
    - Contains the **TypeScript source code** for the project.
    - Specifically, it includes a single file named **'main.ts'**.
    - TypeScript files (*.ts) are where you write your game logic using the Kaboom.js library.
2. **'sounds' Folder**:
    - Contains the **sound files** used in the project.
    - Includes the following sound files:
        - 'dead.mp3': Possibly used for player character death or game over.
        - 'music.mp3': Likely background music during gameplay.
        - 'Pew.mp3': Possibly a sound effect (e.g., shooting, hitting).
        - 'score.mp3': Possibly played when the player scores points.
3. **'sprites' Folder**:
    - Contains the **image files** used for the sprites in the game.
    - Includes the following sprite images:
        - 'bg.png': Background image for the game.
        - 'ghosty.png': Image representing a ghost character.
        - 'mBG.jpg': Possibly another background image.
        - 'shinchan.png': Image representing a character named Shinchan.
        - 'walls.png': Image representing walls or obstacles.
4. **'Js' Folder**:
    - Contains a single file named **'multiplayer.js'**.
    - The purpose of this file is unclear from the context provided. It might be related to multiplayer functionality, but further details are needed.
5. **'template.html' File**:
    - This HTML file serves as the **entry point** for the project.
    - It likely links to the TypeScript and JavaScript files, as well as any necessary CSS.
    - The template.html file defines the structure of the webpage where the game will run.
6. **Other Unspecified Files**:
    - There are additional files at the top level of the directory: 'TS', 'main.ts', '5', and another 'template.html'.
    - Unfortunately, their purpose or meaning remains unclear without additional context.

7.2 – CODING OF THE PROJECT

This code is written in TypeScript involving the Kaboom library for game turn of events. It is a basic 2D game with a labyrinth like level, where the player controls a person and should try not to be hit by foes. The game has a few scenes, including the menu scene, the game scene, and the credits scene.

The game purposes console and contact contribution to move the player character around the screen. At the point when the player character crashes into a slug terminated by a foe, the game closures and the player is returned to the menu scene.

The code incorporates a few capabilities for making game items, for example, buttons and the player character, and for taking care of client input. It additionally incorporates capabilities for stacking game resources, like pictures and sounds, and for making and overseeing game scenes.

The game purposes a tile-based level plan, where each tile is addressed by a sprite and has a particular way of behaving. The level is made utilizing a labyrinth age calculation, which makes an irregular labyrinth like example of walls and open spaces.

The game likewise incorporates a straightforward molecule impact that is played when the player character is hit by a slug. The particles are made utilizing the 'addKaboom' capability, which adds an eruption of particles at a particular situation on the screen.

By and large, this code is a straightforward illustration of a 2D game constructed utilizing the Kaboom library and TypeScript. It incorporates a few normal game improvement ideas, for example, level age, client info, and resource stacking.

BELOW ARE THE CODE SNIPPETS FOR BETTER UNDERSTANDING –

```
import kaboom from "kaboom";
import "kaboom/global";

// Initialize context
kaboom({
  width: window.innerWidth,
  height: window.innerHeight,
  background: [0, 0, 0],
});

const w = window.innerWidth;
const h = window.innerHeight;
```

- Kaboom Module Imported.
- Kaboom Window Initialized With Responsive layout.
- Constants for Width & Height were declared for a standard ratio in later Development.

- All The sprites loaded in the above game.

```
// Load Assets
loadSprite("shinchan", "sprites/shinchan.png");
loadSprite("bg", "sprites/bg.png");
loadSprite("walls", "sprites/walls.png");
loadSprite("ghosty", "sprites/ghosty.png");
loadSprite("mBG", "sprites/mBG.jpg");
loadSound("BG-TUNE", "sounds/ShinChan Naughty BGM.mp3");
loadSound("dead", "sounds/dead.mp3");
loadSound("Pew", "sounds/Pew.mp3");
```

```
function addButton(txt, p, f) {
  // add a parent background object for the button
  const btn = add([
    rect(240, 80, { radius: 8 }),
    pos(p),
    area(),
    scale(1),
    anchor("center"),
    outline(4),
  ]);
  // add a child object that displays the text
  btn.add([
    text(txt),
    anchor("center"),
    color(0, 0, 0),
  ]);
  // onHoverUpdate() comes from area() component
  // it runs every frame when the object is being hovered
  btn.onHoverUpdate(() => {
    const t = time() * 10;
    btn.color = hsl2rgb((t / 10) % 1, 0.6, 0.7);
    btn.scale = vec2(1.2);
    setCursor("pointer");
  });
  // onHoverEnd() comes from area() component
  // it runs once when the object stopped being hovered
  btn.onHoverEnd(() => {
    btn.scale = vec2(1);
    btn.color = rgb();
  });
  // onClick() comes from area() component
  // it runs once when the object is clicked
  btn.onClick(f);
  return btn;
}
```

- ADD BUTTON function declare to create buttons in further.

- Created scene menu.
- Added buttons.

```
scene("menu", () => {
  // reset cursor to default on frame start for easier cursor management
  onUpdate(() => setCursor("default"));
  // Add a background image to the menu scene
  const backgroundImage = add([
    sprite("mBG"), // Replace "mBG" with the actual image asset name
    pos(0, 0),
  ]);
  // Calculate the scaling factors for width and height
  const scaleX = width() / width(backgroundImage);
  const scaleY = height() / height(backgroundImage);
  // Set the scale of the background image to fit the screen
  scale(backgroundImage, scaleX, scaleY);

  addButton("Start", vec2(200, 100), () => startGame());
  addButton("Quit", vec2(200, 300), () => {
    debug.log("bye");
    window.close(); // Try to close the tab
  });
  addButton("Credits", vec2(450, 200), () => showCredits());;
});
function showCredits() {
  go("credits");
}
// Define the startGame function to transition to the "game" scene
function startGame() {
  // Transition to the "game" scene
  go("game");
}
// Start with the "menu" scene
go("menu");
```

```
scene("credits", () => {
  // Add a background image to the credits scene
  const backgroundImage = add([
    sprite("mBG"), // Replace "mBG" with the actual image asset name
    pos(0, 0),
  ]);
  // Calculate the scaling factors for width and height
  const scaleX = width() / width(backgroundImage);
  const scaleY = height() / height(backgroundImage);
  // Set the scale of the background image to fit the screen
  scale(backgroundImage, scaleX, scaleY);

  // Add a text label with the credits
  add([
    text("Game Development: Aryan Kaul\nFaculty Guide: Mr Krashnakant Gupta")
  ]);
  // Add a button to return to the menu
  addButton("Back", vec2(650, 200), () => go("menu"));
});
```

- Defined Scene Credits

BELOW IS THE MAIN GAME SCENE WHERE ALL THE FUNCTIONS AND THE EXECUTION OF THE GAMEPLAY WILL TAKE PLACE

```
// Game scene
scene("game", () => {
```

```
    // BACKGROUND TUNE
    const bgMusic = play("BG-TUNE", {
      volume: 0.1, // Adjust the volume as needed
      loop: true, // Loop the music
    });


    // Define the size of your blocks and map
    let blockSize = 64; // Block size is 64x64 pixels
    let mapWidth = width() / blockSize; // Map width is 1920 pixels
    let mapHeight = height() / blockSize; // Map height is 1080 pixels

    // Fill the map
    for (let x = 0; x < mapWidth; x++) {
      for (let y = 0; y < mapHeight; y++) {
        add([
          sprite("bg"),
          pos(x * blockSize, y * blockSize),
        ]);
      }
    }

    const TILE_WIDTH = 64;
    const TILE_HEIGHT = 64;

    // Adjust the width and height of the maze to fill a 1920x1080 page
    const MAZE_WIDTH = Math.floor(width() / TILE_WIDTH);
    const MAZE_HEIGHT = Math.floor(height() / TILE_HEIGHT);

    function createMazeMap(width, height) {
      const size = width * height;

      function getUnvisitedNeighbours(map, index) {
        const n = [];
        const x = Math.floor(index / width);
        if (x > 1 && map[index - 2] === 2) n.push(index - 2);
        if (x < width - 2 && map[index + 2] === 2) n.push(index + 2);
        if (index >= 2 * width && map[index - 2 * width] === 2) n.push(index - 2
* width);
        if (index < size - 2 * width && map[index + 2 * width] === 2)
n.push(index + 2 * width);
        return n;
      }

      const map = new Array(size).fill(1, 0, size);
```

```
      map.forEach((_, index) => {
        const x = Math.floor(index / width);
        const y = Math.floor(index % width);
        if ((x & 1) === 1 && (y & 1) === 1) {
          map[index] = 2;
        }
      });

      const stack = [];
      const startX = Math.floor(Math.random() * (width - 1)) | 1;
      const startY = Math.floor(Math.random() * (height - 1)) | 1;
      const start = startX + startY * width;
      map[start] = 0;
      stack.push(start);
      while (stack.length) {
        const index = stack.pop();
        const neighbours = getUnvisitedNeighbours(map, index);
        if (neighbours.length > 0) {
          stack.push(index);
          const neighbour = neighbours[Math.floor(neighbours.length *
Math.random())];
          const between = (index + neighbour) / 2;
          map[neighbour] = 0;
          map[between] = 0;
          stack.push(neighbour);
        }
      }

      return map;
    }

    function createMazeLevelMap(width, height, options) {
      const symbols = options?.symbols || {};
      const map = createMazeMap(width, height);
      const space = symbols[" "] || " ";
      const fence = symbols["#"] || "#";
      const detail = [
        space,
        symbols["╸"] || "╸", //  1
        symbols["╹"] || "╹", //  2
        symbols["┘"] || "┘", //  3
        symbols["╺"] || "╺", //  4
        symbols["━"] || "━", //  5
        symbols["┗"] || "┗", //  6
        symbols["┻"] || "┻", //  7
```

```
        symbols["╷"] || "╷", //  8
        symbols["╖"] || "╖", //  9
        symbols["│"] || "│", //  a
        symbols["┤"] || "┤", //  b
        symbols["┌"] || "┌", //  c
        symbols["┬"] || "┬", //  d
        symbols["├"] || "├", //  e
        symbols["┼ "] || "┼ ", //  f
      ];
      const symbolMap = options?.detailed
        ? map.map((s, index) => {
            if (s === 0) return space;
            const x = Math.floor(index % width);
            const leftWall = x > 0 && map[index - 1] == 1 ? 1 : 0;
            const rightWall = x < width - 1 && map[index + 1] == 1 ? 4 : 0;
            const topWall = index >= width && map[index - width] == 1 ? 2 : 0;
            const bottomWall = index < height * width - width && map[index +
width] == 1 ? 8 : 0;
            return detail[leftWall | rightWall | topWall | bottomWall];
          })
        : map.map((s) => {
            return s == 1 ? fence : space;
          });
      const levelMap = [];
      for (let i = 0; i < height; i++) {
        levelMap.push(symbolMap.slice(i * width, i * width + width).join(""));
      }
      return levelMap;
    }

    const level = addLevel(
      createMazeLevelMap(MAZE_WIDTH, MAZE_HEIGHT, {}), // Use the adjusted maze
dimensions
      {
        tileWidth: TILE_WIDTH,
        tileHeight: TILE_HEIGHT,
        tiles: {
          "#": () => [
            sprite("walls"),
            area(),
            body({isStatic: true}),
          ],
        },
      }
    );
```

```
// PLAYER
const SPEED = 300;
const player = add([
  sprite("shinchan"),
  pos(90,80),
  scale(0.098),
  area(),
  body(),
]);

// MOVEMENT CODE

onKeyDown("left", () => {
  player.move(-SPEED, 0);
});

onKeyDown("right", () => {
  player.move(SPEED, 0);
});

onKeyDown("up", () => {
  player.move(0, -SPEED);
});

onKeyDown("down", () => {
  player.move(0, SPEED);
});

onKeyDown("a", () => {
  player.move(-SPEED, 0);
});

onKeyDown("d", () => {
  player.move(SPEED, 0);
});

onKeyDown("w", () => {
  player.move(0, -SPEED);
});

onKeyDown("s", () => {
  player.move(0, SPEED);
});
```

```
onClick(() => {
  player.moveTo(mousePos());
});

// Mobile controls
let touchStartPos = new Vec2(0, 0);

onTouchStart((pos, t) => {
  touchStartPos = pos;
});

onTouchMove((pos, t) => {
  const deltaX = pos.x - touchStartPos.x;
  const deltaY = pos.y - touchStartPos.y;

  // Adjust sensitivity based on your needs
  const touchSensitivity = 8000;

  if (Math.abs(deltaX) > Math.abs(deltaY)) {
    // Horizontal movement
    if (deltaX > touchSensitivity) {
      player.pos.x += SPEED;
    } else if (deltaX < -touchSensitivity) {
      player.pos.x -= SPEED;
    }
  } else {
    // Vertical movement
    if (deltaY > touchSensitivity) {
      player.pos.y += SPEED;
    } else if (deltaY < -touchSensitivity) {
      player.pos.y -= SPEED;
    }
  }

  // Update touchStartPos for the next frame
  touchStartPos = pos;
});




const ENEMY_SPEED = 160;
const BULLET_SPEED = 1200;

const enemy = add([
```

```
    sprite("ghosty"),
    pos(width() - 80, height() - 80),
    anchor("center"),
    // This enemy cycles between 3 states, and starts from the "idle" state
    state("move", ["idle", "attack", "move"]),
  ]);

  // Run the callback once every time we enter the "idle" state.
  // Here we stay "idle" for 0.5 seconds, then enter the "attack" state.
  enemy.onStateEnter("idle", async () => {
    await wait(0.5);
    enemy.enterState("attack");
  });

  function restartButton(txt, p, f) {
    // add a parent background object for the button
    const re = add([
      rect(160, 50, { radius: 8 }),
      pos(p),
      area(),
      scale(1),
      anchor("center"),
      outline(4),
    ]);

    // add a child object that displays the text
    re.add([
      text(txt),
      anchor("center"),
      color(0, 0, 0),
    ]);

    // onHoverUpdate() comes from area() component
    // it runs every frame when the object is being hovered
    re.onHoverUpdate(() => {
      const t = time() * 10;
      re.color = hsl2rgb((t / 10) % 1, 0.6, 0.7);
      re.scale = vec2(1.2);
      setCursor("pointer");
    });

    // onHoverEnd() comes from area() component
    // it runs once when the object stopped being hovered
    re.onHoverEnd(() => {
      re.scale = vec2(1);
```

```
    re.color = rgb();
  });

  // onClick() comes from area() component
  // it runs once when the object is clicked
  re.onClick(f);

  return re;
}

restartButton("Restart", vec2(170, 40), () => {
  location.reload(); // Reload the page
});

enemy.onStateEnter("attack", async () => {
  if (player.exists()) {
    const dir = player.pos.sub(enemy.pos).unit();

    // Play the "Pew" sound on shoot
    play("Pew", {
      volume: 0.3, // Adjust the volume as needed
    });

    add([
      pos(enemy.pos),
      move(dir, BULLET_SPEED),
      rect(12, 12),
      area(),
      offscreen({ destroy: true }),
      anchor("center"),
      color(BLUE),
      "bullet",
    ]);
  }

  await wait(1);
  enemy.enterState("move");
});

enemy.onStateEnter("move", async () => {
  await wait(2);
  enemy.enterState("idle");
});

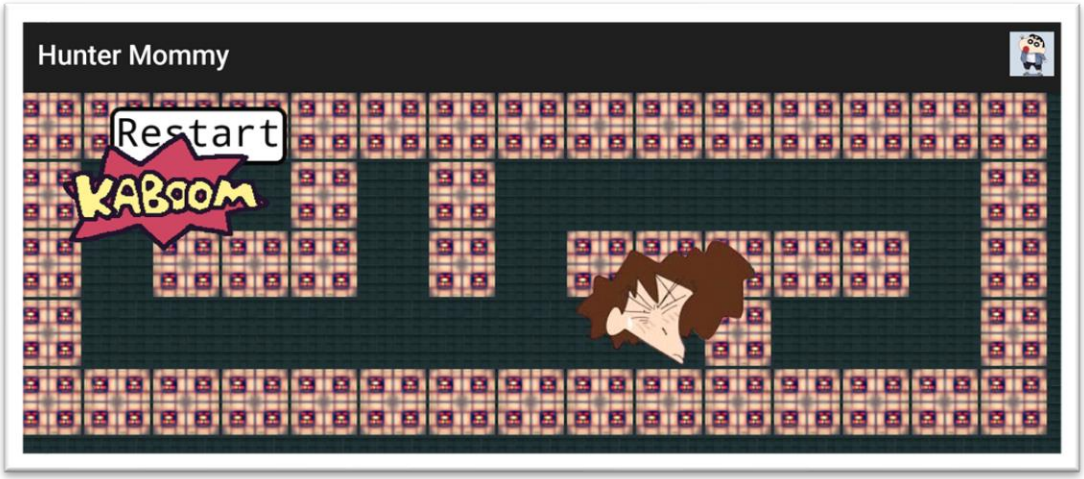
enemy.onStateUpdate("move", () => {
```

```
    if (!player.exists()) return;
    const dir = player.pos.sub(enemy.pos).unit();
    enemy.move(dir.scale(ENEMY_SPEED));
  });


  player.onCollide("bullet", (bullet) => {
    destroy(bullet);
    destroy(player);
    addKaboom(bullet.pos);
    bgMusic.stop();
    play("dead", {
      volume: 0.3, // Adjust the volume as needed
    });
  });

})
```

CHAPTER 9: SNAPSHOTS



CHAPTER 10: CONCLUSION

Kaboom.js stands apart as a convincing game improvement library because of its easy to use plan and strong capacities. Its linguistic structure, described by straightforwardness and lucidity, engages engineers at all ability levels to make an interpretation of innovative ideas into intelligent games without any problem. The complete documentation going with Kaboom.js fills in as a significant asset, offering clear direction and guides to speed up the educational experience.

One of the champion elements of Kaboom.js is its adaptability. Engineers can outfit its ability to make many games, from straightforward models to complex, staggered undertakings. The system's particular design empowers simple joining of different parts, adding to a more smoothed out improvement work process.

Besides, the dynamic Kaboom.js people group assumes a significant part in its prosperity. Customary updates, cooperative conversations, and shared assets inside the local area improve the general advancement experience. This feeling of kinship encourages a climate where designers can look for help, share experiences, and aggregately add to the improvement of the system.

As a JavaScript library, Kaboom.js adjusts consistently with contemporary web improvement rehearses. Its similarity with web advancements guarantees that engineers can use their current information and abilities, settling on it an alluring decision for those all around acquainted with JavaScript.

Fundamentally, Kaboom.js arises as a system for game improvement as well as a steady environment that energizes inventiveness, cooperation, and effectiveness. With its instinctive nature, broad documentation, and solid local area backing, Kaboom.js shows what itself can do as a champion device for making dazzling and intelligent gaming encounters.

CHAPTER 10: REFFERENCES

- https://www.freecodecamp.org/news/learn-javascript-game-development-full-course/
- How to Get Started with Game Development? - GeeksforGeeks
- What Is Game Development? (freecodecamp.org)
- Kaboom.js (kaboomjs.com)
- Download the latest indie games - itch.io
- Space Shooter with Kaboom.js | Replit Docs
- Game Tutorial (w3schools.com)

2% Overall Similarity

Top sources found in the following databases:

- 1% Internet database
- Crossref database
- 1% Submitted Works database
- 0% Publications database
- Crossref Posted Content database

TOP SOURCES

The sources with the highest number of matches within the submission. Overlapping sources will not be displayed.

1	**Napier University on 2013-12-05** Submitted works	1%
2	**dokumen.pub** Internet	1%