Project Summary

Ghost Blade

by: Alisher Zhussip & Norair Tarasov

CP3407

project choice: Game

Introduction

In our new game, players take on the role of Takeshi, a hero on a journey through a world invaded by strange forces. The game offers exciting combat and a gripping story, as Takeshi fights enemies like wolves and bandits. With two starting levels full of action and exploration, the game is inspired by titles like *Sekiro* and *Samurai Girl*, mixing fast combat with a deep story.

Target Audience

Our game is perfect for:

- **Action-Adventure Fans:** Players who enjoy exciting combat and exploring new places.
- **Story Lovers:** Gamers who appreciate strong storytelling and character growth.
- **Fans of Japanese Culture:** Those interested in samurai legends and Japanese settings.
- **Sekiro and Samurai Girl Fans:** People who liked the combat and stories in these games.

Goal of Our Project

The main goal of our project is to create a fun and immersive action-adventure game that gives players a memorable experience. We want to:

- **Deliver Exciting Combat:** Offer a challenging and enjoyable combat system that keeps players interested.
- **Tell a Great Story:** Create a story that grabs players' attention and makes them want to know more.
- **Build a Big World:** Develop a detailed, immersive game world that encourages players to explore.

What the Game is About

Takeshi's journey starts during what seems like a normal camping trip. Suddenly, he sees a huge ship come down from the sky, marking the arrival of powerful beings from another dimension. Curious and worried, Takeshi rushes to the boat, only to find that it has brought creatures and vessels corrupting the land. Players will guide Takeshi through different levels, fighting these new threats, uncovering the mysteries behind their arrival, and working to bring balance back to the world.

Why We Are Making This

We are making this game because we believe in the power of storytelling through interactive experiences. Our team is passionate about:

- **Creating Unique Experiences:** Giving players a mix of challenging combat and deep storytelling.
- **Exploring New Worlds:** Designing interesting worlds that players can get lost in.
- **Innovating in Game Design:** Pushing the limits of traditional action-adventure games by adding unique mechanics and story elements.

What We Are Planning to Do with It

Our plans for the game include:

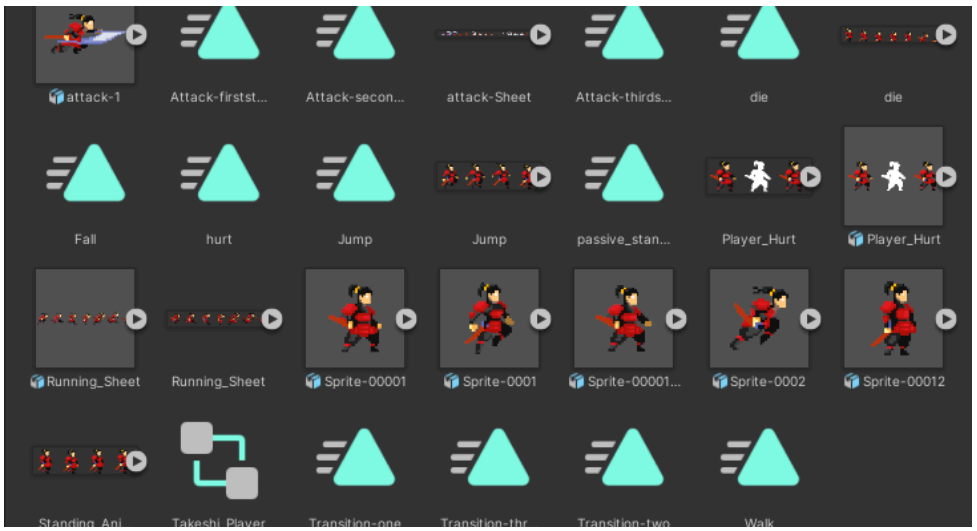
- **Expanding Levels:** Adding more levels to extend the story and offer greater challenges.
- **Improving Mechanics:** Continuously enhancing combat mechanics and gameplay based on player feedback.
- **Engaging the Community:** Building a community of players who can share their experiences and help shape the game's development.
- **Future Updates and Content:** Regularly releasing updates, new content, and possible expansions to keep the game fresh and exciting.

A tool that we were using for our game

We use Aseprite for creating all the game assets because it offers a range of features tailored specifically for pixel art and animation. Aseprite is renowned for its precision in pixel manipulation, allowing us to control each pixel accurately, which is crucial for high-quality 2D game graphics.

One of its standout features is its powerful animation tools. Aseprite supports frame-by-frame animation, onion skinning, and sprite sheet management, which are essential for creating smooth and dynamic animations. The application also provides advanced layer management, making it easier to organize and edit multiple layers of assets efficiently.

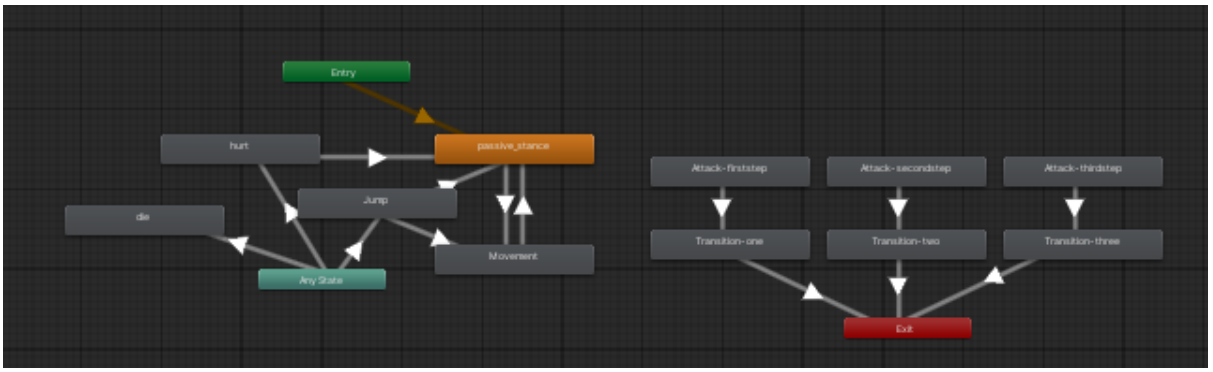
Additionally, Aseprite allows for custom color palettes, helping us maintain consistent color schemes across different assets and levels. It also includes tools for designing and editing tilemaps, which are useful for constructing game environments and levels. The user-friendly interface and customizable workspace further streamline our workflow, making the process of creating detailed and animated pixel art more efficient and enjoyable.



We have organized all animations for Takeshi into a single directory, which we then use directly in our game. Every animation, from idle states to dynamic attack sequences, was meticulously created by our team to ensure a cohesive and unique visual style.

By consolidating these animations into one directory, we streamline the process of managing and accessing them within Unity. The Unity Animator tool plays a crucial role in this setup, allowing us to manage and control Takeshi’s animations effectively. The Animator tool provides a visual interface where we can define transitions, blend animations, and fine-tune animation parameters, ensuring that Takeshi's movements and actions are both fluid and responsive.

This organized approach not only simplifies animation management but also enhances our ability to maintain consistency and quality across Takeshi’s character animations, contributing to a more polished and engaging gameplay experience.



All animations for Takeshi are managed through the Unity Animator tool, where each animation is triggered based on specific rules and conditions we've set. This setup allows us to control which animations play and when, according to the game's logic and the character's actions.

For example, animations for running, jumping, and attacking are activated based on player input and in-game events. By configuring these rules within the Animator tool, we ensure that Takeshi's movements and behaviors respond accurately to gameplay scenarios, creating a seamless and dynamic animation experience. This approach not only enhances the character's realism but also streamlines the animation workflow, making it easier to adjust and refine animations as needed.

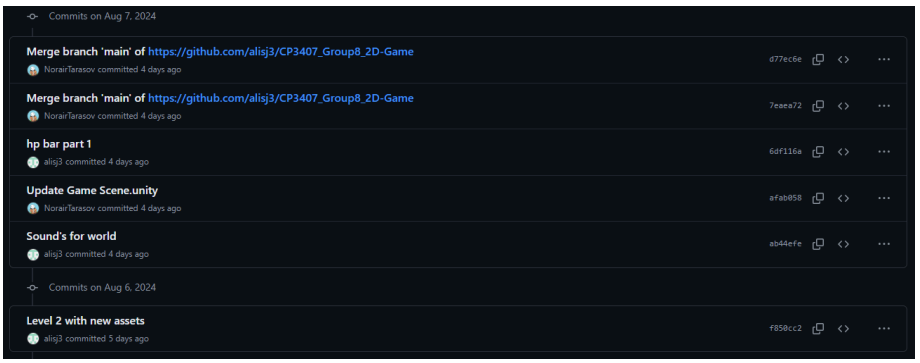
GitHub

We utilized GitHub Desktop for version control throughout the development of our game. This tool provided a seamless and intuitive experience, especially for team members who were less familiar with Git's command-line interface. The drag-and-drop functionality, visual representation of changes, and easy branch management significantly streamlined our workflow. GitHub Desktop also made collaboration more efficient by simplifying the process of syncing changes with the remote repository and enabling quick access to repository history and diffs.

In addition to these benefits, GitHub Desktop's built-in conflict resolution tool was invaluable in maintaining code integrity when multiple team members were working on the same files. The ability to easily revert changes and manage pull requests directly within the app further enhanced our productivity.

To ensure our Unity project was well-organized and optimized, we created a `.gitignore` file. This file helped us exclude unnecessary files and folders, such as build outputs, temporary files, and local settings, from being tracked in version control. By doing so, we kept our repository clean and focused only on the essential assets and scripts, which helped reduce clutter and avoid potential conflicts.

For our project we made a lot of commits into github where you can see them.



Piece of commits in GitHub

Actual game Review

The game begins with a cyberpunk-themed main menu, featuring three primary options: "Play," "Settings," and "Exit."

When you select "Play," you'll be taken directly into the game, beginning the story of Takeshi. This option immediately launches the gameplay, immersing you in the dark and gritty world of the samurai/cyberpunk setting. (Picture 1)

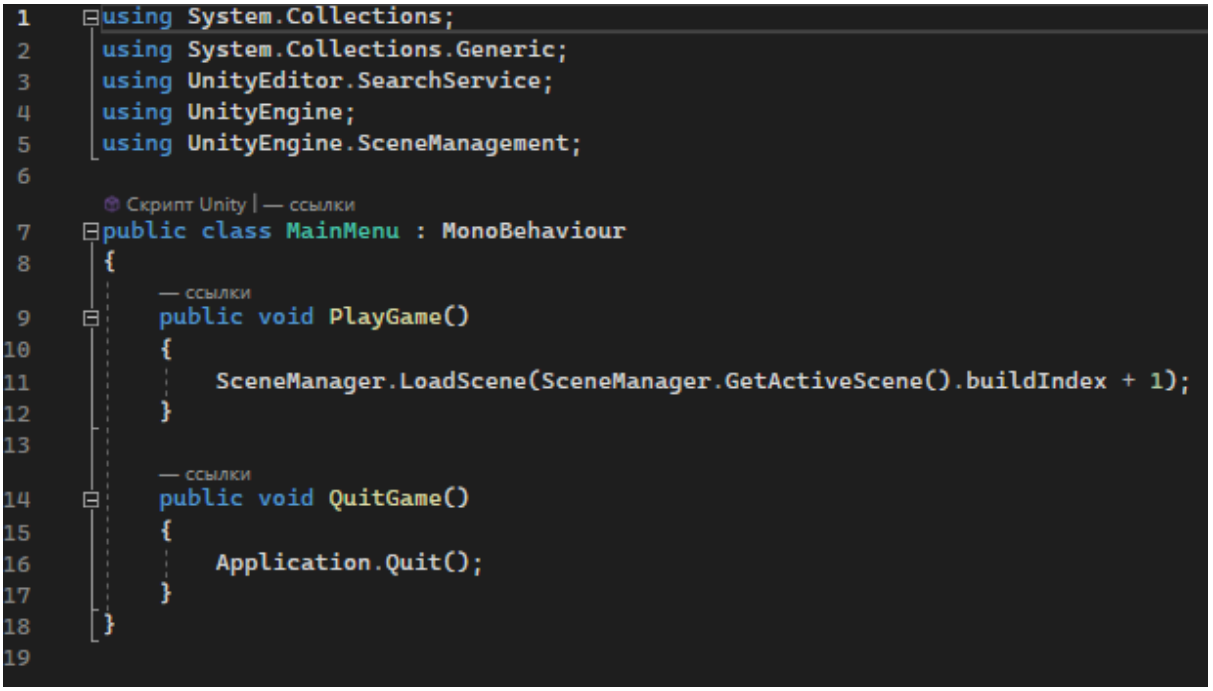
Picture 1. Main Menu

Choosing "Settings" allows you to customize your gaming experience. Here, you can adjust various options such as sound tailoring the game to suit your preferences and ensuring optimal performance on your device.

Finally, the "Exit" option lets you quit the game, taking you back to your desktop or home screen. This choice provides a quick and straightforward way to close the game when you're finished playing.

In our main menu, we have a simple script that triggers the game scene, allowing players to start the actual gameplay. In Unity Engine, each window or screen is managed as its scene, which provides a structured way to organize and control different aspects of the game. (Picture 2)

By using separate scenes for different parts of the game, we maintain a clean and modular approach to development. This method makes it easier to manage and update individual sections of the game without affecting others, ensuring a smoother workflow and a more stable final product.



```
using System.Collections;
using System.Collections.Generic;
using UnityEditor.SearchService;
using UnityEngine;
using UnityEngine.SceneManagement;

Скрипт Unity | — ссылки
public class MainMenu : MonoBehaviour
{
    — ссылки
    public void PlayGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
    }

    — ссылки
    public void QuitGame()
    {
        Application.Quit();
    }
}
```

Picture 2. Scene Management

After pressing "Play," the game launches with the first cutscene, where Takeshi witnesses a spaceship plummeting from the sky. This dramatic moment sets the tone for the story, immersing you in the chaos and intrigue of Takeshi's world as he stands in awe, watching the vessel descend. This event marks the beginning of his journey, drawing you deeper into the cyberpunk narrative.

Once the cutscene concludes, the player takes control and can begin playing the game. The first location is a dense and dangerous forest, where Takeshi must navigate through treacherous terrain to reach the crashed spaceship. The forest is filled with hazards that challenge Takeshi’s survival skills, making this a critical part of the journey.

In the top left corner of the screen, a health bar is displayed, allowing you to monitor Takeshi's health as he battles through the forest. This visual cue is essential for managing resources and planning strategies to ensure that Takeshi makes it to the spaceship safely. (Picture 3)

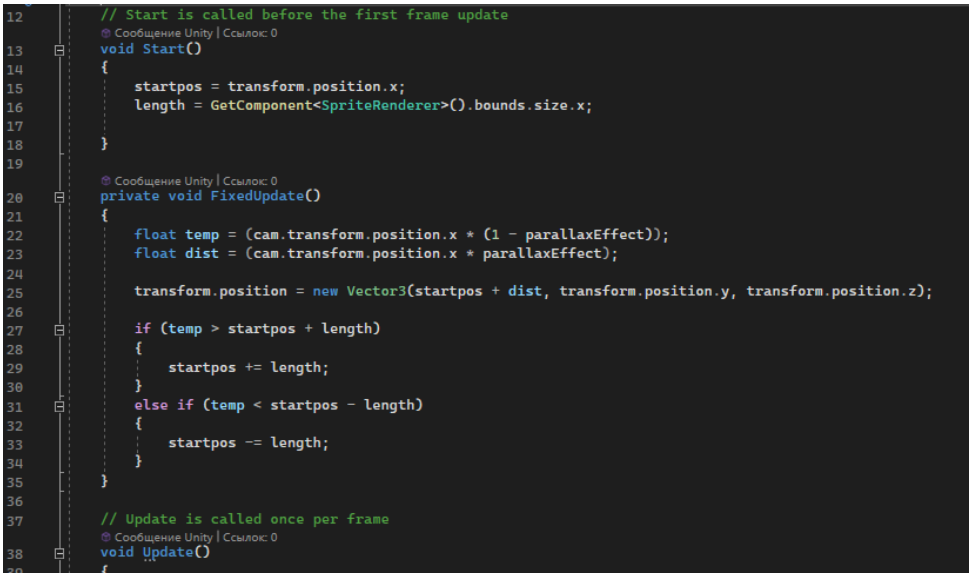
Picture 3. Game Scene

The player can move Takeshi left and right, jump, perform a series of different attacks, and dash in the direction he's facing. You have the option to control these movements using either the W, A, S, or D keys or the arrow keys on the keyboard, giving you flexibility in how you play.

All of Takeshi’s animations and attack sequences were meticulously drawn by our team. We didn't use any external assets for the main character, ensuring a unique and original experience that reflects our vision for the game.

In the new level, you'll experience a parallax animated background that moves in sync with the player. This technique creates a sense of depth and immersion, giving the game world a more dynamic and three-dimensional appearance. As the player navigates the Cyberpunk-themed environment, the layered background enhances the physicality of the world, making the setting feel more expansive and alive.

The `Parallax_Background` script creates a parallax scrolling effect for backgrounds, enhancing the depth and immersion of the game world. It adjusts the background's position relative to the camera's movement, making the background scroll at a different rate than the foreground. The script is initialized by calculating the background's width and starting position. During each update, it modifies the background's position based on the camera's x-coordinate and a defined parallax effect. It also handles repositioning the background seamlessly to ensure continuous scrolling, which helps maintain a dynamic and engaging visual experience.



```
12      // Start is called before the first frame update
        Сообщение Unity | Ссылок: 0
13      void Start()
14      {
15          startpos = transform.position.x;
16          length = GetComponent<SpriteRenderer>().bounds.size.x;
17
18      }
19
        Сообщение Unity | Ссылок: 0
20      private void FixedUpdate()
21      {
22          float temp = (cam.transform.position.x * (1 - parallaxEffect));
23          float dist = (cam.transform.position.x * parallaxEffect);
24
25          transform.position = new Vector3(startpos + dist, transform.position.y, transform.position.z);
26
27          if (temp > startpos + length)
28          {
29              startpos += length;
30          }
31          else if (temp < startpos - length)
32          {
33              startpos -= length;
34          }
35      }
36
37      // Update is called once per frame
        Сообщение Unity | Ссылок: 0
38      void Update()
```

Picture 4. Parallax Background Script Piece

Main player scripts explanation:

The `PlayerMovement` script controls a 2D character's movement, including running, jumping, and dashing, while managing input and animations. Key variables like `horizontal move` track player input, and `running speed` determine movement speed. The `is facing right` boolean tracks the character's orientation, while the `can jump` controls jumping.

The script also includes a dash mechanic with variables like `canDash` and `isDashing` to manage dash availability and execution. The `Dash` coroutine temporarily disables gravity, applies dash velocity, and creates a visual trail. The `FixedUpdate` method handles physics-based movement, while the `Update` method captures real-time input.

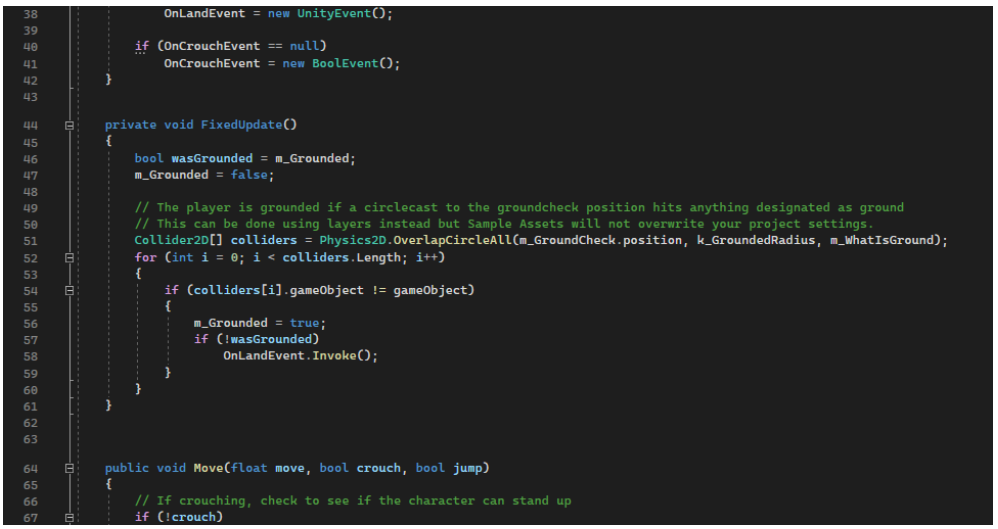
Additionally, an `Animator` component controls animations, and a `Flip` method (currently commented out) is intended to flip the character's sprite when changing direction. Overall, the script ensures smooth and responsive character control for a 2D game. (Picture 4)

```
36
37      // Update is called once per frame
        Сообщение Unity | Ссылок: 0
38      void Update()
39      {
40          if (isDashing)
41          {
42              return;
43          }
44
45          horizontalMove = Input.GetAxisRaw("Horizontal");
46          animator.SetFloat("Speed", Mathf.Abs(horizontalMove));
47
48          if (Input.GetKeyDown(KeyCode.Space))
49          {
50              canJump = true;
51              animator.SetBool("isJumping", true);
52          }
53
54          if (Input.GetKeyDown(KeyCode.LeftShift) && canDash)
55          {
56              StartCoroutine(Dash());
57          }
58
59          /*   Flip();*/
60      }
61
        Ссылок: 0
62      public void OnLanding()
63      {
64          animator.SetBool("isJumping", false);
65      }
```

Picture 5. Piece of Player movement Script

You can see the full script for movement in Scripts/Player Scripts/PlayerMovement.

Following the `PlayerMovement` script, we also have a script dedicated to controlling various aspects of character behavior, including air control and crouching. This script allows us to fine-tune how the character moves and interacts with the environment, whether they're on the ground or in the air. It provides flexibility in adjusting movement mechanics, making it easier to customize the gameplay experience to match the desired feel of the game. (Picture 5)



```
            OnLandEvent = new UnityEvent();

        if (OnCrouchEvent == null)
            OnCrouchEvent = new BoolEvent();
    }


    private void FixedUpdate()
    {
        bool wasGrounded = m_Grounded;
        m_Grounded = false;

        // The player is grounded if a circlecast to the groundcheck position hits anything designated as ground
        // This can be done using layers instead but Sample Assets will not overwrite your project settings.
        Collider2D[] colliders = Physics2D.OverlapCircleAll(m_GroundCheck.position, k_GroundedRadius, m_WhatIsGround);
        for (int i = 0; i < colliders.Length; i++)
        {
            if (colliders[i].gameObject != gameObject)
            {
                m_Grounded = true;
                if (!wasGrounded)
                    OnLandEvent.Invoke();
            }
        }
    }


    public void Move(float move, bool crouch, bool jump)
    {
        // If crouching, check to see if the character can stand up
        if (!crouch)
```

Picture 6. Player Controllers script

The `CharacterController2D` script manages a 2D character's movement, including walking, jumping, and crouching. Key variables include `m_JumpForce` for jumping strength, `m_CrouchSpeed` for speed while crouching, and `m_MovementSmoothing` for smooth movement transitions. It also features `m_AirControl` to allow steering in the air.

Collision detection is handled using `m_GroundCheck` and `m_CeilingCheck` to determine if the player is grounded or if there’s a ceiling overhead. `m_CrouchDisableCollider` is used to disable a collider when crouching. The script tracks if the player is grounded and whether they are facing right.

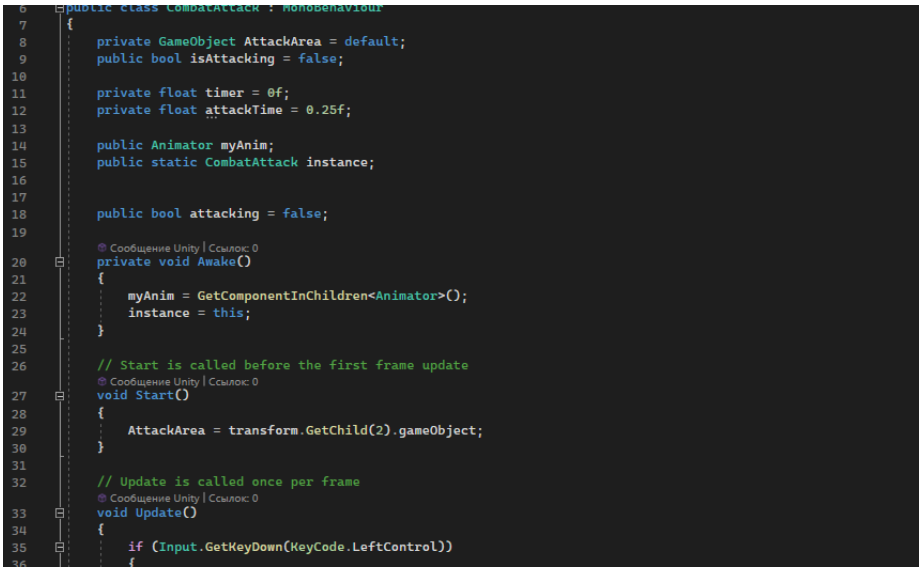
The `Awake` method initializes components and event handlers. In `FixedUpdate`, it checks if the player is on the ground and triggers `OnLandEvent` if the player lands. The `Move` method handles movement, crouching, and jumping, adjusting speed, and applying forces as needed. The `Flip` method flips the character’s sprite based on movement direction.

Overall, the script provides detailed control over character movement and state management, ensuring smooth and responsive gameplay.

The `CombatAttack` script controls the player's attack actions in the game. It manages an `AttackArea` GameObject, which represents the zone affected by the player's attack.

Key functions include handling input to trigger attacks, timing the attack duration, and managing the visibility of the `AttackArea`. When the left control key is pressed, the script activates the attack by setting `attacking` to true and showing the `AttackArea`. The script then uses a timer to deactivate the attack area after the attack time has elapsed.

Overall, the `CombatAttack` script ensures that attack actions are visually represented and managed efficiently, coordinating the attack's timing and activation. (Picture 7)



```
public class CombatAttack : MonoBehaviour
{
    private GameObject AttackArea = default;
    public bool isAttacking = false;

    private float timer = 0f;
    private float attackTime = 0.25f;

    public Animator myAnim;
    public static CombatAttack instance;


    public bool attacking = false;

    Сообщение Unity | Ссылок: 0
    private void Awake()
    {
        myAnim = GetComponentInChildren<Animator>();
        instance = this;
    }

    // Start is called before the first frame update
    Сообщение Unity | Ссылок: 0
    void Start()
    {
        AttackArea = transform.GetChild(2).gameObject;
    }

    // Update is called once per frame
    Сообщение Unity | Ссылок: 0
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.LeftControl))
        {
```

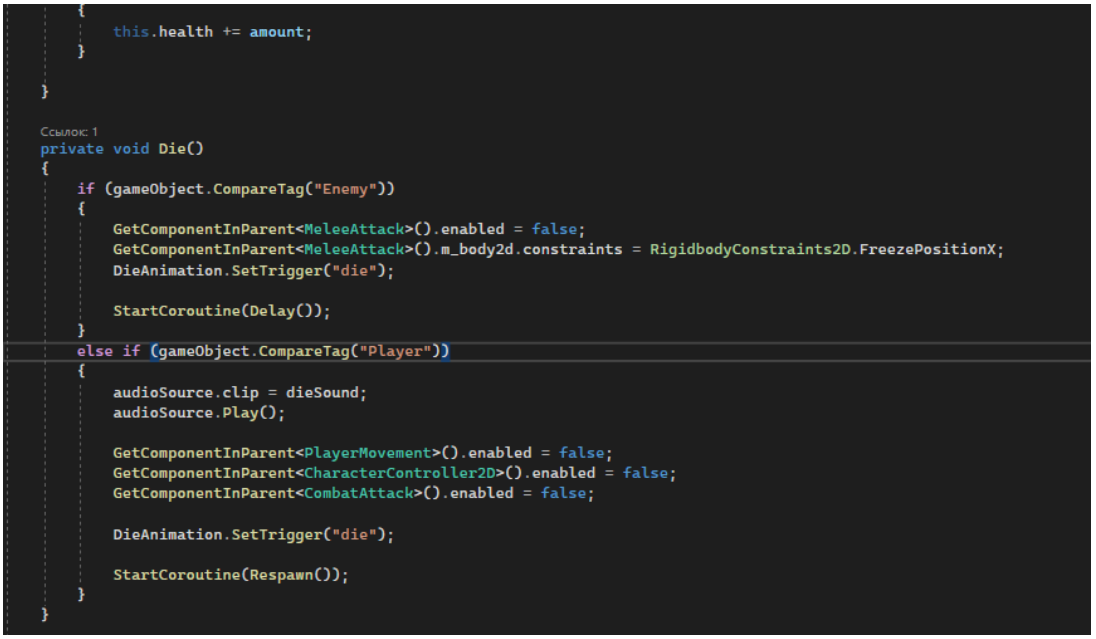
Picture 7. Player Attack Script

Also, we implemented the script that can be used on every enemy or player that is responsible for the health of the object and taking the damage from other objects.

The `PlayerHealth` script manages health for both player and enemy characters. It tracks health, updates health bars, and handles damage, healing, and death.

Key functions include reducing health when damaged, updating the health bar, and triggering animations and sound effects. When health reaches zero, the script handles the death process: for

enemies, it disables attack components and destroys the object; for players, it disables movement and combat components, plays a death sound, and respawns by loading a new scene after a delay. The script also includes methods for healing and adjusting the health display. (Picture 8)



```
        {
            this.health += amount;
        }

    }

    Ссылок: 1
    private void Die()
    {
        if (gameObject.CompareTag("Enemy"))
        {
            GetComponentInParent<MeleeAttack>().enabled = false;
            GetComponentInParent<MeleeAttack>().m_body2d.constraints = RigidbodyConstraints2D.FreezePositionX;
            DieAnimation.SetTrigger("die");

            StartCoroutine(Delay());
        }
        else if (gameObject.CompareTag("Player"))
        {
            audioSource.clip = dieSound;
            audioSource.Play();

            GetComponentInParent<PlayerMovement>().enabled = false;
            GetComponentInParent<CharacterController2D>().enabled = false;
            GetComponentInParent<CombatAttack>().enabled = false;

            DieAnimation.SetTrigger("die");

            StartCoroutine(Respawn());
        }
    }
```

Picture 8 . health script

Afterward, we have another unique script that is responsible for finding the player on the map and attacking him on a given range. Everything can be modified by the developers in another window.

The `MeleeAttack` script controls an enemy's melee combat. It manages attack cooldowns, detects the player, and handles movement and animations.

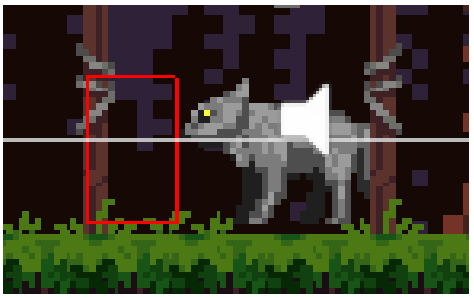
The script checks if the player is within the detection range and moves towards them. It updates the enemy's sprite direction and triggers attack animations when in range. The `PlayerInSight` method uses a `BoxCast` to determine if the player is within attack range and updates the player's health if detected.

Overall, the script coordinates enemy movement, attack actions, and animations, ensuring the enemy responds effectively to player presence and attack opportunities. (Picture 9)

```
96    private bool PlayerInSight()
97    {
98        RaycastHit2D hit =
99            Physics2D.BoxCast(boxCollider.bounds.center + transform.right * range * transform.localScale.x * colliderDistance,
100           new Vector3(boxCollider.bounds.size.x, boxCollider.bounds.size.y, boxCollider.bounds.size.z),
101           0, Vector2.left, 0, playerLayer);
102
103       if (hit.collider != null)
104         playerHealth = hit.transform.GetComponent<PlayerHealth>();
105
106
107
108       return hit.collider != null;
109
110   }
111
112
113
      Сообщение Unity | Ссылок: 0
114   private void OnDrawGizmos()
115   {
116       Gizmos.color = Color.red;
117       Gizmos.DrawWireCube(boxCollider.bounds.center + transform.right * range * transform.localScale.x * colliderDistance,
118           new Vector3(boxCollider.bounds.size.x * range, boxCollider.bounds.size.y, boxCollider.bounds.size.z));
119   }
120
```

Picture 9. Piece of MeeleAttack script for Enemies

Enemies will hit the player on a given range and detect them on a given range. The red box in front of the enemy is the range attack of the enemy.



Picture 10. Enemy

We can manage every variable and give it other values at any time by using this window. (Picture 11)

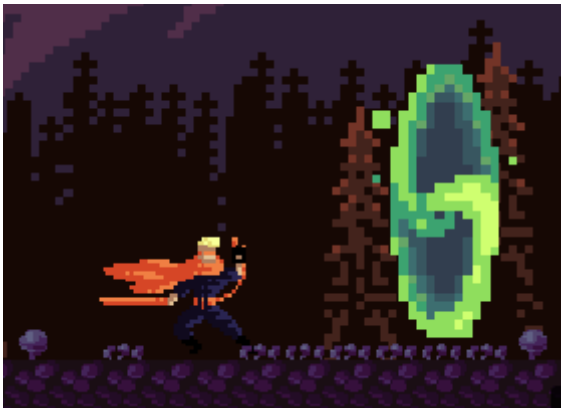
Picture 11. Enemy Script Modify

Our initial enemies in the game are the wolf and the bandit. Thanks to the scripts we've developed, we can easily create any enemy type by simply attaching these scripts to objects with animations. This flexibility allows us to adapt the enemy behavior and characteristics without needing to rewrite code for each new type. The versatility of this script is a testament to the extensive time and effort we invested in its development and implementation. It represents a core component of our game, enabling dynamic and varied enemy interactions that enhance the overall gameplay experience.

Picture 12. Bandit

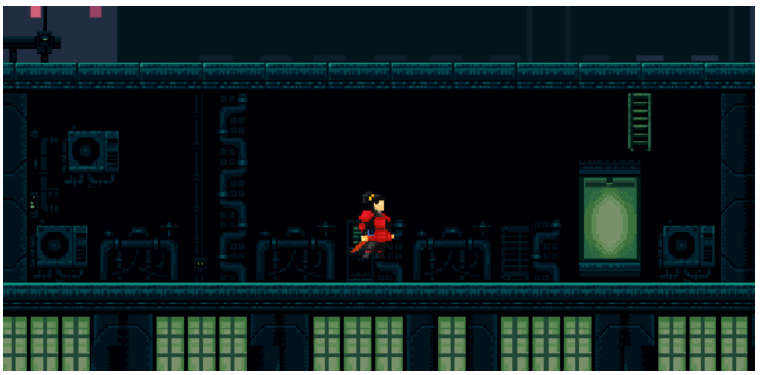
After the level, a cutscene unfolds featuring a time traveler stepping into a portal. Takeshi, driven by curiosity or pursuit, follows him into the portal. This scene sets the stage for the next

chapter of the adventure, signaling a shift in the storyline and potentially introducing new challenges and environments. The transition to this new sequence not only advances the plot but also deepens the player's engagement with Takeshi's journey, hinting at the exciting developments and encounters that lie ahead. (Picture 13)



Picture 13. Time Traveler and portal

Following the cutscene, you will enter a new level featuring a Cyberpunk-themed map. This level immerses you in a futuristic, neon-lit cityscape, complete with high-tech visuals and a gritty, urban atmosphere. The shift in environment not only enhances the game's aesthetic diversity but also introduces new challenges and elements that align with the Cyberpunk setting, further enriching Takeshi's adventure. (Picture 14)



Picture 14. level 2

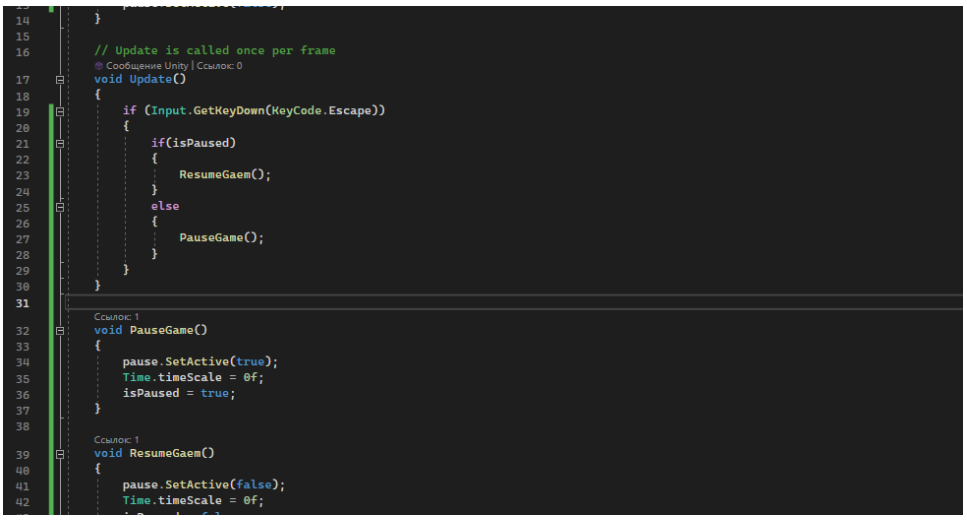
As a final feature of our game, players have the option to pause the game. While paused, you can choose to either quit the game entirely or restart the current level. This feature provides flexibility and control, allowing players to manage their gameplay experience and address any interruptions or challenges they encounter. (Picture 15)

Picture 15. Pause

The `Pause` script enables the pause functionality in the game, allowing players to pause and resume gameplay with ease. Upon starting, it hides the pause menu by setting its associated `GameObject` to inactive. When the Escape key is pressed, the script toggles the pause state.

If the game is currently paused, it will call the `ResumeGame` method, which hides the pause menu and resumes the game by setting `Time.timeScale` back to 1. Conversely, if the game is not paused, it will call the `PauseGame` method, which activates the pause menu and stops the game by setting `Time.timeScale` to 0.

Additionally, the script includes a `QuitGame` method that allows players to exit the game application entirely. This feature provides players with control over their gameplay experience, allowing them to pause, resume, or quit as needed. (Picture 17)



```
    }

    // Update is called once per frame
    Сообщение Unity | Ссылок: 0
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            if(isPaused)
            {
                ResumeGaem();
            }
            else
            {
                PauseGame();
            }
        }
    }

    Ссылок: 1
    void PauseGame()
    {
        pause.SetActive(true);
        Time.timeScale = 0f;
        isPaused = true;
    }

    Ссылок: 1
    void ResumeGaem()
    {
        pause.SetActive(false);
        Time.timeScale = 0f;
```

Picture 17. Pause the Game script Piece

Burndown Chart

In our first iteration, our primary focus was on implementing visual elements for the player. This involved a significant amount of drawing, designing, and integrating these assets into our project. Each team member contributed to the creation of various visual components, ensuring they were aligned with our game's aesthetic and thematic goals. As we completed each task, we committed our work to the project repository, progressively building out the player's visual experience.

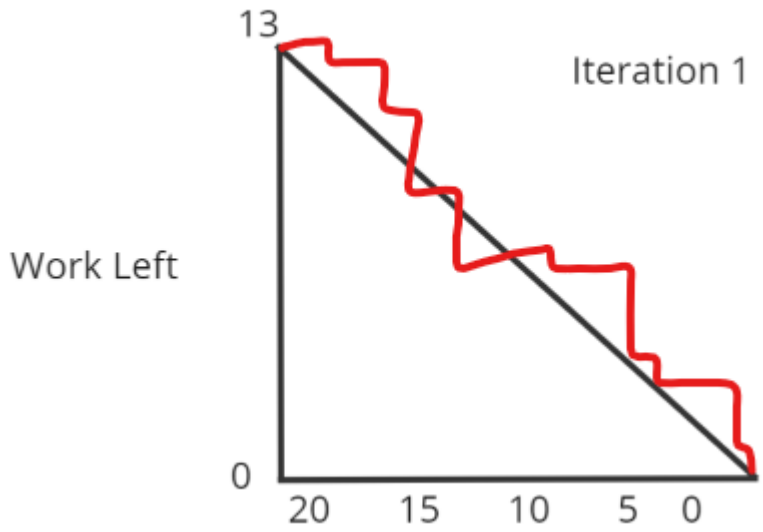
The burndown chart you've provided visualizes the progress we made during this iteration. The chart tracks the amount of work left (on the y-axis) against the number of days remaining (on the x-axis).

- The black diagonal line represents the ideal progress, where work decreases consistently and steadily each day until completion.

- The red line represents our actual progress. As seen, our work fluctuated—some days were more productive, resulting in a steeper decline, while others saw slower progress with only a slight reduction in the workload.

Despite the occasional variability in our pace, we managed to complete the necessary tasks by the end of the iteration. This chart reflects the challenges of managing creative work like drawing and

design, where some tasks may take longer than expected, but ultimately, we were able to deliver everything on time. (Picture 18)



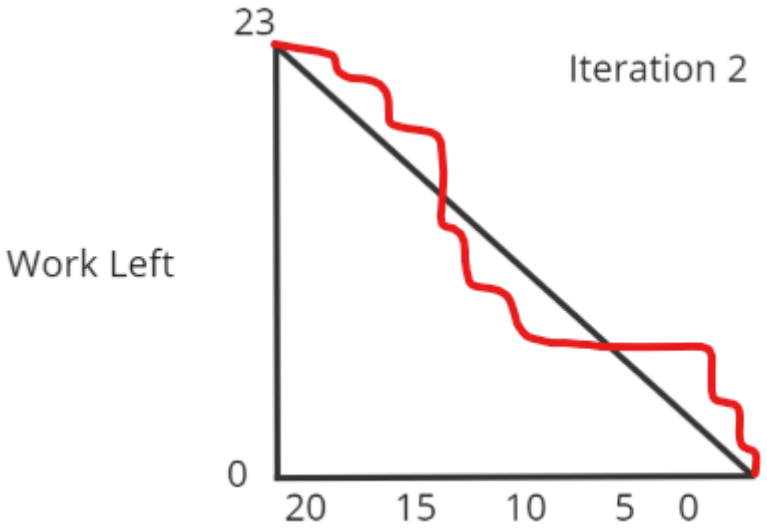
Picture 18. Iteration 1 chart

In our second iteration, we shifted focus from visual elements to core gameplay mechanics. Specifically, we implemented player movement, attack functionalities, and various interactions between the player and enemies. Additionally, we started developing the game map, ensuring that the player's actions would be smoothly integrated within the game world.

The burndown chart for this iteration shows how our work progressed over time. Despite encountering some early challenges, which may have been due to the complexity of the tasks, we managed to gain momentum as the iteration progressed. The red line in the chart reflects this, showing a gradual decrease in the amount of work left as we completed each task.

By the end of the iteration, we successfully completed all planned tasks, ensuring that the player mechanics and interactions were fully functional and integrated into the game. This chart

highlights our ability to overcome initial obstacles and achieve our goals within the given timeframe. (Picture 19)



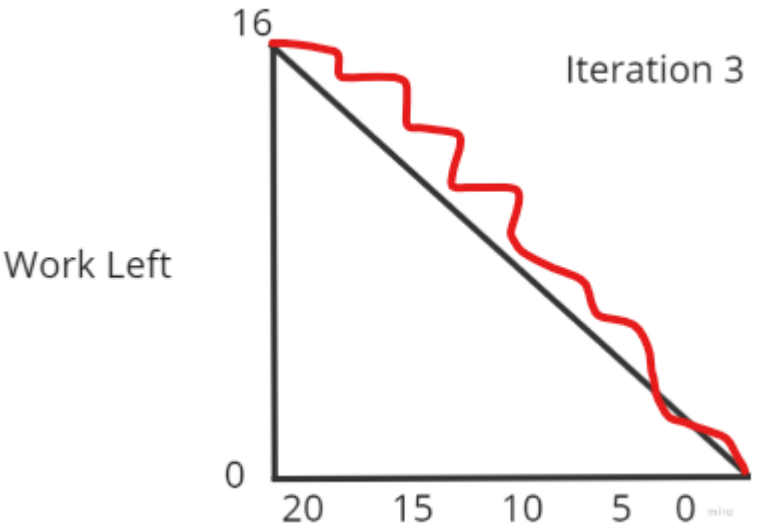
Picture 19. Iteration 2

In iteration 3, we dedicated a significant amount of time to optimizing our game and refining various scripts. This phase was crucial as we wanted to establish a strong foundation before advancing to more complex features and content. The work involved meticulous debugging, performance enhancements, and code refactoring to ensure that our game ran smoothly and efficiently.

We focused on improving the responsiveness of player controls, reducing lag during combat, and streamlining interactions with the environment. Additionally, we revisited earlier implementations to correct any issues that had arisen during previous iterations. This included fine-tuning enemy AI, optimizing collision detection, and ensuring that the map's design facilitated seamless gameplay.

Moreover, we enhanced the game's overall architecture by organizing scripts more effectively, improving modularity, and removing any redundant code. This careful optimization and correction process not only improved the game's performance but also made future development more manageable and less prone to errors. (Picture 20)

By the end of iteration 3, we had solidified a robust and well-optimized base for our game. This provided us with the confidence to move forward with the development of new features, knowing that our core systems were stable and efficient.



Picture 20. Iteration 3

Development Environment

Our primary development platform will be Unity, chosen for its robust and comprehensive toolset tailored for 2D game development. Unity offers advanced features such as sprite management, physics simulations, and animation tools, all of which will be essential for bringing our game to life. Its cross-platform capabilities will also allow us to develop and test the game on various devices, ensuring a broad audience reach upon release.

Programming Languages

C# will be our main programming language, which seamlessly integrates with Unity's powerful scripting API. This choice enables us to efficiently implement game mechanics, player controls, and AI behaviors. By using C#, we can take full advantage of object-oriented programming principles, leading to cleaner and more maintainable code. Additionally, C#'s extensive community

support and documentation will provide valuable resources as we tackle complex challenges during development.

Source Code Repositories/Version Control

For version control and collaboration, we will use GitHub, which offers a secure and private repository to protect our codebase. GitHub's version control features will allow us to track changes, manage branches, and collaborate seamlessly with our team members, ensuring that everyone is on the same page and contributing to the project's success. The integration of GitHub Desktop will further streamline our workflow by providing a user-friendly interface for committing, merging, and resolving conflicts.

Project Collaboration Tools

To enhance project management and communication, we will employ tools like Trello or Jira. These platforms will help us track tasks, manage sprints, and prioritize features, ensuring that the project stays on schedule and that team members are always aware of their responsibilities. Clear communication and task visibility will be maintained through these tools, which will be vital for coordinating efforts across different aspects of the game development process.

Development Tools

Visual Studio will serve as our integrated development environment (IDE), offering a powerful and versatile platform for writing, testing, and debugging our C# code. Its advanced features, such as IntelliSense and integrated debugging, will accelerate our development process and help identify and resolve issues quickly. For creating and refining game sprites, we will use tools like Photoshop or Aseprite, which provide the necessary capabilities for high-quality 2D art and animations. These tools will be integral to crafting the visual aesthetic of our game, ensuring that it meets the artistic vision we have set out to achieve.

Testing and Continuous Integration

To ensure the stability and quality of our game, we will integrate automated testing and continuous integration (CI) practices into our development process. Tools like Unity Test Runner will

be used to create unit tests for game mechanics, while CI platforms such as GitHub Actions will automate the testing and build processes. This approach will allow us to catch bugs early, maintain code quality, and ensure that every new feature is properly tested before being merged into the main branch.

Conclusion

Ghost Blade is designed to give players an immersive action-adventure experience, packed with intense combat, a deep story, and a fascinating world to explore. Inspired by games like *Sekiro* and *Samurai Girl*, our goal is to create something that fans of the genre will love while adding our unique touch. We're excited to keep expanding the game, connecting with our community, and delivering an unforgettable adventure.