Akdeniz Eğitim Araştırmaları Dergisi, 2025, Cilt 19 Sayı 51, Sayfa 01-104
Mediterranean Journal of Educational Research 2025, Volume 19 Issue 51, Page 01-104

Copyright © 2025. Açık Erişim Makale CC BY-NC-ND

Environmentally Friendly 3D Game Developed Based on Design-Based Research Method

Şenay Kocakoyun Aydoğan[1]

Abstract: Today, environmental problems have become a global concern. It is now an inevitable necessity to be aware of our responsibility as humanity and to search for solutions. At this point, the educational game created in this study aims to raise awareness about garbage sorting and recycling for a sustainable life. This game is more than just a 3D game, it aims to equip players with environmental awareness. The gameplay mechanics are based on correctly sorting various types of garbage. Plastic, paper and other materials provide a game experience that requires players to think strategically. In this direction, an educational game design has been developed to increase interest in a sustainable experience centered on learning. In this study, the game development process was based on the design-based research method. Unity 3D game engine, which is popularly used by game developers, was used. Unity 3D is preferred due to its high number of users and free usage. In the game, it is emphasized how critical it is for our environment and future generations to separate waste such as plastic, paper, glass, bags, etc. correctly. In addition to entertaining the players, the game aims to raise their awareness of environmental responsibility and increase their participation in the recycling movement. With the codes shared, it is expected to contribute in terms of guiding those who will start developing games using the Unity 3D engine or prepare simulation environments in interdisciplinary fields.

Keywords: Environment, Recycling, Sustainability, Responsibility, Educational game, Unity 3D

Geliş Tarihi: 05.10.2024 – **Kabul Tarihi:** 27.03.2024 – **Yayın Tarihi:** 30.03.2025

DOI:

INTRODUCTION

Games are basically explained by the elements of struggle, imagination, curiosity and control (Malone, & Lepper, 2021). Malone (1980) emphasizes that with these elements, games present the player with elements of struggle to overcome in a virtual environment that the player actively experiences and controls and arouse cognitive and sensory curiosity with its changing dynamics. Educational games, on the other hand, are games that are used to achieve specific learning goals and provide immersive learning experiences with the components they contain (De Freitas, 2006). With their unique rules, they offer individuals the opportunity for social

[1] **Şenay Kocakoyun Aydoğan,** İstanbul Gedik University, Department of Information Security Technology, ORCID: 0000-0002-3405-6497. Email**:** senay.aydogan@gedik.edu.tr

interaction by providing active experiences in meaningful and realistic environments (Gee, 2003).

Research consistently supports the educational potential of games, highlighting their effectiveness in improving learning outcomes (Dodlinger, 2007; Tüzün, Yılmaz-Soylu, Karakuş, İnal, & Kızılkaya, 2009; Sánchez & Olivares, 2011). Additionally, they foster enjoyable learning environments (Ebner & Holzinger, 2007; Bressler & Bodzin, 2013), maintain sustained interest (Liao, Chen, Cheng, Chen, & Chan, 2011), facilitate collaborative learning (Wong, Hsu, Sun, & Boticki, 2013), enhance engagement (Schwabe & Göth, 2005), and support the development of critical 21st-century skills such as creativity, communication, collaboration, information, and ICT literacy (Dodlinger, 2007; Sourmelis, Ioannou, & Zaphiris, 2017).

Further evidence from extensive analyses of empirical studies reinforces these findings. Connolly, Boyle, MacArthur, Hainey, and Boyle (2012), examining 129 experimental studies, concluded that games significantly impact individuals' perceptual, cognitive, behavioral, affective, and motivational domains. Games are particularly noted for their ability to motivate learners effectively (Dodlinger, 2007; Huizenga, Admiral, Akkerman, & Dam, 2009; Papastergiou, 2009), positioning them as powerful educational tools (Oblinger, 2004). Similarly, Hainey, Connolly, Boyle, Wilson, and Razak (2016), through an analysis of 105 studies, emphasized the positive influence of digital educational games on affective and motivational outcomes, behavioral change, cooperation and communication skills, perceptual-cognitive development, and knowledge acquisition. In alignment with these insights, educational games have significant potential in addressing contemporary global issues, such as environmental awareness. Environmental awareness encompasses developing a sense of responsibility and understanding toward the natural environment, including key issues like nature conservation, sustainability, waste management, energy conservation, and water resource protection. The significance of environmental awareness has increased in recent years due to escalating environmental crises and the rapid depletion of natural resources worldwide (Kiziroğlu, 2023). This awareness is critical not only for preserving natural resources but also for safeguarding human health and welfare. Environmental pollution adversely affects vital resources such as air, water, and soil, leading to detrimental consequences for human health. Additionally, ensuring environmental sustainability provides future generations with a livable and sustainable planet (Esen & Esen, 2018).

As societies grapple with urgent environmental issues, the search for practical solutions has become increasingly vital. In this context, the educational game we have developed aims specifically at raising awareness about waste sorting and recycling practices. By merging entertainment with education, the game endeavors to impart meaningful environmental knowledge and foster sustainable behaviors among players. Going beyond mere gameplay, the primary objective of the game is to cultivate and reinforce environmental consciousness among players. Gameplay mechanics center on accurately sorting various types of waste, including plastics, paper, and other materials. Through strategically designed interactive challenges, the game mirrors real-world recycling scenarios, providing players with practical skills and insights into responsible environmental practices. Consequently, the game not only promises an engaging and enjoyable user experience but also encourages active participation in recycling initiatives. It seeks to inspire players to adopt environmentally responsible behaviors, demonstrating how individual actions, when multiplied, can significantly impact community-level environmental health. Ultimately, this educational gaming approach exemplifies how integrating learning, and entertainment can effectively promote meaningful, long-lasting behavioral changes and heightened environmental awareness.

Purpose and Importance of the Study

The aim of the game is not only to entertain but also to educate players about the importance and practices of garbage sorting through engaging educational components. Emphasizing the critical role of waste management-including plastic, paper, glass, and bags-the game seeks to foster awareness and encourage sustainable environmental responsibility. By integrating learning and entertainment, the study aims to cultivate environmentally responsible behaviors that can positively impact future generations.

METHODOLOGY

This study, which aims to model the story of the game in the development process and transform it into practice, was carried out using the design-based research method. In design-based research method, it is aimed to develop theories, phenomena or practices that are thought to affect or explain the learning-teaching process in its natural environment from a cyclical perspective (Barab & Squire, 2004). Design-based research can result in practical solutions that aim to develop theoretical perspectives, products, processes, programs or policies for the

learning-teaching process (McKenney & Reeves, 2018). In this study, it was aimed to develop a game with a design suitable for the intended educational purposes.

Game Development Environments

To translate educational goals into practical game elements, appropriate game development tools must be utilized. A game engine, a vital component in this process, refers to the software—either paid or free—used by developers and companies to create games (Tuğtekin & Kaleci, 2011). Game engines provide numerous predefined library files containing functions, classes, and other resources developed in various programming languages. By leveraging these libraries, game developers can significantly streamline their workflow, eliminating redundant coding tasks and accelerating the development process

3D Game Design

The basis of the computer game development phase is the selection of a game engine suitable for the structure of the type of game to be developed. In the decision-making phase at the beginning of the game development process, the purpose and genre of the game should be decided and appropriate methods should be followed. The game development process is carried out in two different stages: basic and advanced. In the simple development process shown in Figure 1, models, skins, sound files and video files are combined in the chapter editor with the codes prepared in the programming language of the game engine, the chapters are prepared and supported by graphical codes, and the work is brought to the completion stage.

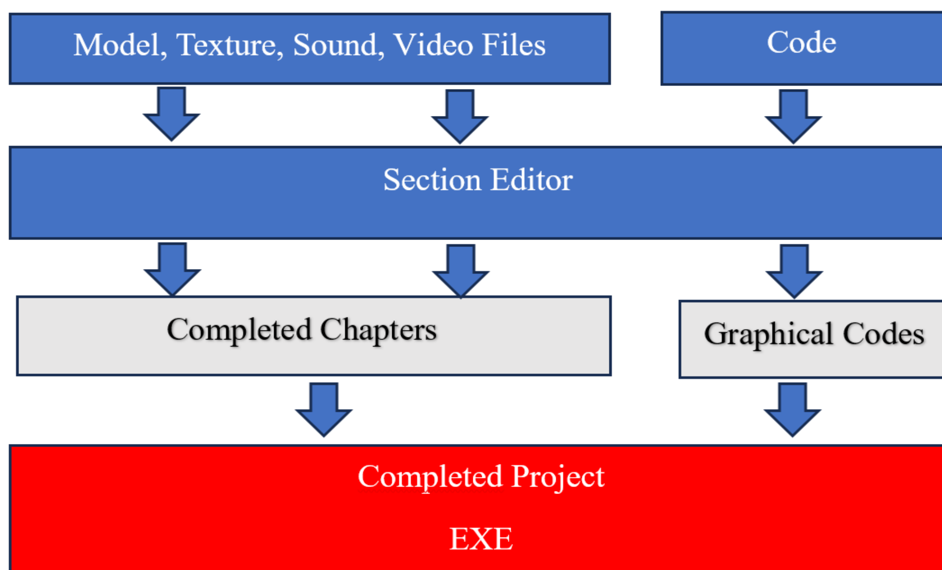

Figure 1. Stages of game development at a basic level (Tuğtekin, & Kaleci, 2011)

On the other hand, the advanced development stage, depicted in Figure 2, involves detailed preparation and customization of models, textures, and comprehensive code libraries. Editors specific to chapters, models, skins, and code are employed, enabling developers to implement intricate graphical and system-level codes. Additionally, the process includes creating or incorporating external libraries to extend the capabilities of the game engine, offering enhanced customization options.

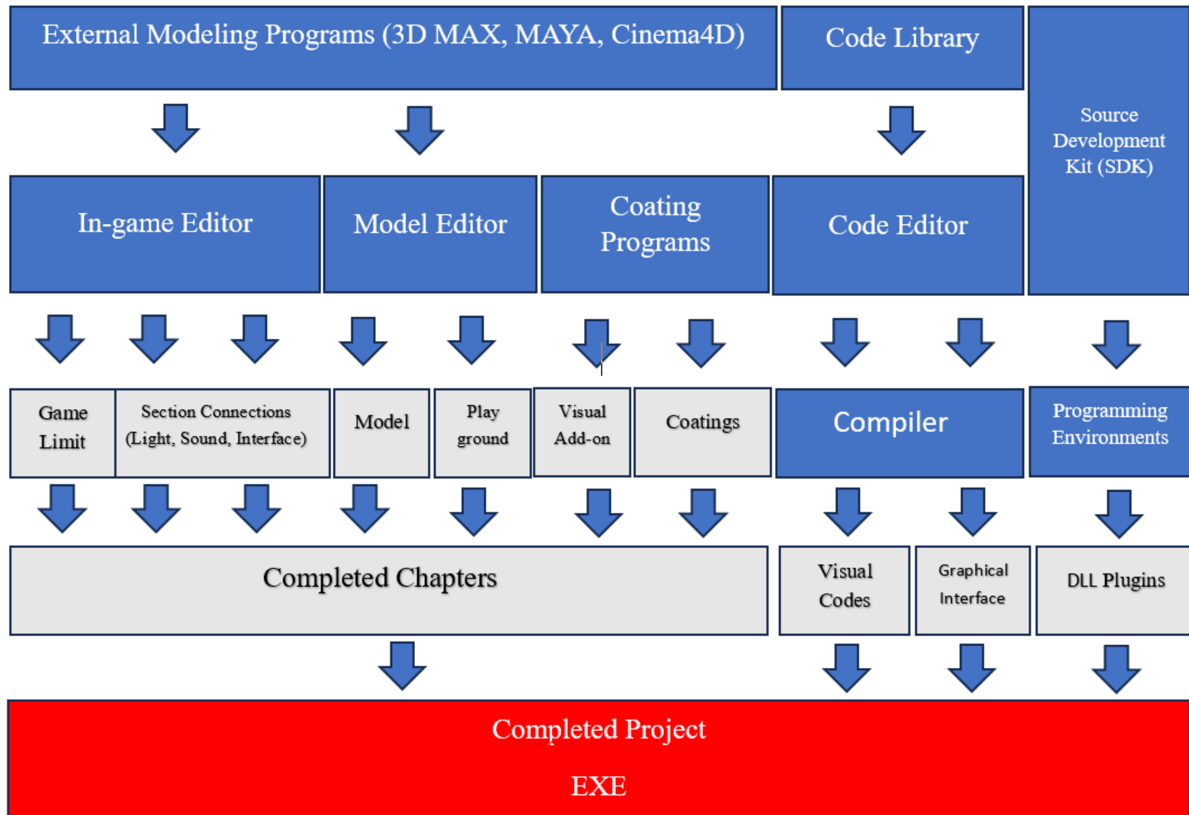

Figure 2. Advanced game development stages (Tuğtekin, & Kaleci, 2011)

In this study, Unity 3D game engine and the programs it contains were used. In the first stage of the study, the locations in the game were designed and appropriate models were created. MED (Model Editor), which the game engine contains, was used to prepare the 3D environments of the modeling in accordance with the reality and the type of the game. After deciding on the game genre, the environments to be modeled were photographed and transferred to digital media, and these environments were used as 3D models. In-game sections were created by combining the 3D models created. In order to use all these structures in the game, the desired environments were created by assigning action and material functions to the models in the sections created by using the programming languages supported by the game engine. In

this study, it is aimed to create a game that has high playability and can work stably in a wide range of systems. For this purpose, only the way of using the modeling system was changed.

Unity 3D Game Engine

The developed educational game is based on the Unity 3D game engine for its successful realization. Unity 3D stands out as a flexible and powerful game engine preferred by many developers today (Jackson, 2015; Okita, 2019). This engine simplifies the programming process of game developers by supporting the C# programming language. UNITY allows games to reach a wider audience thanks to its ability to publish games on different platforms (Unity Technologies, 2005). For this reason, Unity is a popular game engine used by many major game companies and independent developers. It is also a game engine used by educational institutions and students, helping young game developers gain important experience before starting their careers. In the rest of the study, the main features of Unity 3D and the values it adds to the game development process are explained.

1.User friendly interface and rapid prototyping

Unity 3D significantly facilitates rapid prototyping due to its intuitive and user-friendly interface. The drag-and-drop functionality simplifies complex development tasks, allowing developers to effortlessly add components, adjust materials, and design animations within scenes (Bond, 2014). This feature minimizes the technical barriers typically encountered in game development, enabling developers to experiment and iterate quickly. As a result, creativity becomes the central focus, empowering developers to swiftly test ideas, refine gameplay mechanics, and accelerate overall game development.

2. 2D and 3D Game Development

Unity 3D offers a wide set of tools for developing both 2D and 3D games. This versatility gives developers the flexibility to create different types of games. For the environmental awareness game developed in this study, Unity 3D provided a wide range of freedom in meeting the requirements of the project, enabling effective design and management of the 3D world.

Unity game engine is a game engine developed with the C# programming language that creates 2D and 3D graphics and applies animation, physics simulation, realistic lighting, etc. (Watson, Voloh, Thomas, Hasan, Womelsdorf, 2019). The Unity game engine provides environments that help subjects to observe or explore. Using this environment in navigation or

stimulus context studies benefits researchers (Jangraw, Johri, Gribetz, & Sajda, 2014). It is seen that Unity is preferred by game developers because it is used by large communities and the number of experienced people is high, the steps to be followed are specified in detail in the Unity documentation, sample codes are shared, and it is free for individual users (Güneş, 2021).

3. Physics and Animation Systems

Modern video games have evolved significantly in terms of visual fidelity, interactivity, and realism. One of the key components of this evolution is the implementation of realistic physics and animation systems. Players expect game characters and objects to move and behave naturally, just like in real life. Unity 3D provides advanced tools, including a powerful physics engine and an Animator Controller system, to create fluid animations and natural physics-based interactions (Tykoski, 2022). These systems significantly enhance gameplay immersion and help create a believable virtual environment.

4. Support for Various Platforms

One of the most significant advantages of Unity 3D is its ability to support multiple platforms efficiently. Game developers can create a game once and deploy it across various devices and operating systems with minimal modifications. This cross-platform compatibility makes Unity a preferred choice for both indie developers and large game studios, as it allows games to reach a broader audience and maximize market potential (Lavieri, 2018).

5. Community and Resources

One of Unity 3D's greatest strengths is its large and active community, which provides invaluable support to developers of all skill levels. Whether a beginner or an experienced professional, developers can benefit from a vast network of users who share knowledge, offer solutions, and collaborate on projects. This extensive community fosters continuous learning and innovation, making Unity an accessible and developer-friendly engine.

In addition to community support, Unity offers a wealth of resources and tools to streamline the development process. The Unity Asset Store is one of the most valuable platforms available to developers, allowing them to integrate ready-made assets, plugins, and tools into their projects with ease (Blackman, 2013; Lavieri, 2018). This significantly reduces development time and effort, enabling teams to focus more on creativity and gameplay mechanics rather than building everything from scratch. The availability of professional-grade

assets—ranging from 3D models and animations to sound effects and AI solutions—empowers developers to enhance their games efficiently and cost-effectively.

Beyond the Asset Store, Unity provides comprehensive documentation, official tutorials, and an active developer forum, all of which contribute to a rich learning ecosystem. Additionally, platforms like Unity Learn, YouTube tutorials, and third-party courses make it easier for aspiring game developers to master the engine and refine their skills. By leveraging Unity's strong community and extensive resources, developers can overcome challenges, optimize their workflow, and bring their creative visions to life. As a result, Unity 3D not only serves as a powerful game engine but also as a collaborative and educational platform that empowers developers to push the boundaries of game design.

Preparation of the Application

For the environmentally friendly 3D game, sketches were made on paper using the story draft, designer or game developer's interests and abilities, indicating the game boundaries, object or area descriptions and environment designs. In order to ensure that the game planned in the educational game development process has realistic features, the objects to be used in the game should be 3D. In this study, the player models to be integrated into the Unity 3D game engine were prepared on 3D models using ready-made graphics programs. Figure 3 shows the 3D models of male and female players.

Figure 3. 3D Player Models

For the environment design, vehicles and material models to be integrated into the Unity 3D game engine, access to https://3dwarehouse.sketchup.com/ was provided and changes were made on the 3D models using graphics programs. Figure 4 shows the 3D models of vehicles, houses and streets used in the environment design.

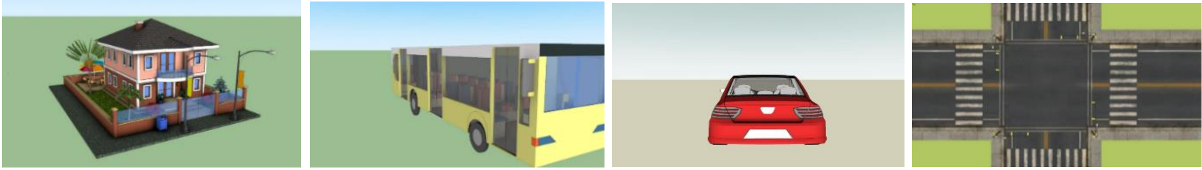

Figure 4. 3D Environment Design, Vehicles and Materials

The introduction and tutorial interface view of the game is shown in Figure 5.

Figure 5. Game login and tutorial interface view

In this small city, bins are not only reserved for general waste, but also for recycling. Collecting materials such as plastic, paper and metal in separate bins is an important step in contributing to the recycling process. In addition, the environmental benefits of recycling are emphasized with informative signs placed in the bins.

Figure 6. Example Garbage sorting screen

This game offers a great opportunity to develop environmental awareness and raise awareness about recycling. All players have to do is to put the garbage in the correct recycling bins. If they make a wrong choice, the garbage will return to its original position and a big cross will appear on the screen and a sound will be heard to indicate that it is wrong. However, if they choose the correct recycling bin, the garbage will disappear and a big check mark will appear on the screen and a sound will be heard to indicate that it is correct. The game will continue in this way until the garbage is gone. Figure 6 shows an example garbage sorting screen. With these steps, the city not only presented a clean image, but also became a model community with environmentally friendly garbage management.

Using the libraries provided by the Unity 3D game engine, we tried to simulate the streets of today. In addition, separate garbage bins were placed for each different type of garbage. It is aimed that people will take care of separate garbage bins for various garbage by relying on their sensitivity on this issue in real life.

Each part up to this stage refers to the design process. This stage is the part where models are transferred to the game environment, area arrangements are made, scene designs are specified, animations are added to players and objects, garbage is separated with indicator symbols, random behavior feature is provided to players using Navigation Mesh, the artificial intelligence tool of Unity 3D, coding in C# language is realized, tested and the educational game process is completed with updates. In addition to these, the codes required for the development of an environmentally friendly Unity 3D application are given in Figure 6.

In game development, character movement and control mechanics are fundamental elements that define player experience. The Unity C# script in Figure 7 is designed to manage a character’s movement, camera interaction, and various gameplay functionalities.

```
0 references
public class PlayerController : MonoBehaviour
{
    #region Public Variables
    1 reference
    public bool CanMove { get; private set; } = true;
    #endregion

    #region Private Variables
    14 references
    private Camera playerCamera;
    15 references
    private CharacterController characterController;
    9 references
    private Vector3 moveDirection;
    3 references
    private Vector2 currentInput;
    4 references
    private float rotationX = 0;

    [Header("Functional Options")]
    1 reference
    [SerializeField] private bool canSprint = true;
    1 reference
    [SerializeField] private bool canJump = true;
    1 reference
    [SerializeField] private bool canCrouch = true;
    1 reference
    [SerializeField] private bool canUseHeadbob = true;
    1 reference
    [SerializeField] private bool willSlideOnSlopes = true;

    [Header("Controls")]
    1 reference
    [SerializeField] private KeyCode sprintKey = KeyCode.LeftShift;
    1 reference
    [SerializeField] private KeyCode jumpKey = KeyCode.Space;
    1 reference
    [SerializeField] private KeyCode crouchKey = KeyCode.LeftControl;
    2 references
    [SerializeField] private KeyCode interactKey = KeyCode.E;
```

Figure 7. Code for controlling the game character

This script plays a crucial role in ensuring smooth and responsive controls, allowing the player to interact seamlessly with the game world.

Public Variables

Public variables are defined within a class and can be accessed externally. These variables allow interaction with the script from other components or settings without modifying the core logic.

- CanMove: Determines whether the character can move or not. This property is accessible externally but is read-only, ensuring that movement restrictions can be imposed dynamically while keeping its value protected from modification.

Private Variables

Unlike public variables, private variables can only be accessed within the class where they are defined. These variables handle essential gameplay mechanics and internal calculations that dictate how the character behaves in response to player input.

- playerCamera: Represents the camera object, ensuring the character's perspective follows player movement.
- characterController: Manages the physical movement of the character using Unity's CharacterController component.
- moveDirection: A vector that determines the direction of the character's movement based on input.
- currentInput: Stores the latest player input data to process movement and actions.
- rotationX: Defines the camera's rotation angle on the X-axis, allowing for smooth and natural-looking vertical movement.

By managing these variables internally, the script maintains clean and organized logic, keeping unnecessary access restrictions to prevent unintended modifications from external sources.

Functional Options

The script includes several toggleable features that customize character movement and interaction, making it adaptable to different gameplay styles.

- canSprint: Enables or disables sprinting functionality, allowing the character to move at a faster pace.
- canJump: Determines whether the character can perform jumping actions.
- canCrouch: Controls whether the character can crouch, adding a stealth or tactical movement mechanic.
- canUseHeadbob: Enables or disables head bobbing, a visual effect that simulates natural head movement when walking or running.
- willSlideOnSlopes: Defines whether the character should slide when traversing steep slopes, adding realism to terrain interaction.

These functional options allow developers to fine-tune movement mechanics and customize character behavior without rewriting core logic, making the script highly flexible for different game genres.

Control Bindings

To ensure smooth and intuitive player interactions, the script defines key bindings for essential movement functions.

- sprintKey: Specifies the sprint button (Default: Left Shift).
- jumpKey: Sets the key used for jumping (Default: Space Key).
- crouchKey: Assigns the crouch key (Default: Left Ctrl).
- interactKey: Defines the key for interacting with objects (Default: E Key).

By allowing custom key mappings, the script ensures that players can adjust controls according to their preferences, improving accessibility and comfort during gameplay.

These codes manage the basic movement mechanics and control options of the character in a game developed using the Unity3D game engine. It is used to control various situations within the game and to allow the player to react to inputs. Through structured code design and modular functionality, it allows developers to create smooth, engaging, and adaptable gameplay within the Unity3D engine.

The parameters in Figure 8 further refine the script by defining the physical properties of the player character, including movement speed, jumping force, and head movement mechanics. These settings ensure that the character interacts naturally with the game environment, providing realistic physics-based interactions.

```
[Header("Movement Parameters")]
2 references
[SerializeField] private float walkSpeed = 3.0f;
2 references
[SerializeField] private float sprintSpeed = 6.0f;
2 references
[SerializeField] private float crouchSpeed = 1.5f;
1 reference
[SerializeField] private float slopeSpeed = 8f;

[Header("Look Parameters")]
1 reference
[SerializeField, Range(1, 10)] private float lookSpeedX = 2.0f;
1 reference
[SerializeField, Range(1, 10)] private float lookSpeedY = 2.0f;
1 reference
[SerializeField, Range(1, 180)] private float upperLookLimit = 80.0f;
1 reference
[SerializeField, Range(1, 180)] private float lowerLookLimit = 80.0f;

[Header("Jumping Parameters")]
1 reference
[SerializeField] private float jumpForce = 8.0f;
1 reference
[SerializeField] private float gravity = 30.0f;

[Header("Crouch Parameters")]
1 reference
[SerializeField] private float crouchHeight = 0.5f;
1 reference
[SerializeField] private float standingHeight = 2f;
3 references
[SerializeField] private float timeToCrouch = 0.25f;
1 reference
[SerializeField] private Vector3 crouchingCenter = new Vector3(0, 0.5f, 0);
1 reference
[SerializeField] private Vector3 standingCenter = new Vector3(0, 0, 0);
9 references
private bool isCrouching;
3 references
private bool duringCrouchAnimation;
```

Figure 8. Parameters designed to control players' physical characteristics

In a first-person perspective game, precise control over movement, looking mechanics, jumping, and crouching is essential for creating an immersive and responsive player experience. The parameters defined in this Unity script provide fine-tuned control over these mechanics, allowing developers to adjust movement speed, camera sensitivity, gravity, and animation behaviors dynamically. By configuring these settings, the game ensures fluid and realistic character movement, enhancing both player immersion and gameplay responsiveness.

Movement Parameters

Movement speed plays a crucial role in defining how the character interacts with the game environment. These parameters allow developers to set different speeds for various movement states:

- walkSpeed: Defines the standard walking speed of the character, ensuring a balanced and natural pace.
- sprintSpeed: Determines the speed when the character is sprinting, enabling faster traversal in open areas.

- crouchSpeed: Specifies the movement speed while crouching, which is typically slower for stealth-based gameplay.
- slopeSpeed: Adjusts movement speed when the character is navigating sloped surfaces, ensuring smooth transitions on uneven terrain.

These settings dictate the overall pacing of movement, allowing developers to fine-tune the balance between realism and gameplay efficiency. For example, a stealth game may require slower crouch speeds, while a fast-paced shooter may benefit from higher sprint speeds for quick maneuvers.

Look Parameters

Smooth and responsive camera movement is essential for first-person games, as it directly influences the player's field of view and aiming precision. The following parameters govern how the player can rotate and adjust their viewpoint:

- lookSpeedX: Controls the horizontal camera sensitivity, determining how fast the player can look left or right.
- lookSpeedY: Controls the vertical camera sensitivity, affecting how quickly the player can look up or down.
- upperLookLimit: Sets the maximum angle the player can look upward, preventing unnatural or excessive rotation.
- lowerLookLimit: Defines the lowest point the player can look downward, ensuring a realistic range of vision.

By fine-tuning these values, developers can create an optimized and comfortable camera system, whether for smooth exploration or precise aiming in combat scenarios.

Jumping Parameters

Jump mechanics add an extra layer of movement flexibility, allowing players to overcome obstacles and navigate varied environments. These parameters dictate the physics behind jumping behavior:

- jumpForce: Determines the height and power of a jump, directly affecting the character’s ability to traverse gaps or obstacles.
- gravity: Governs the force pulling the character down after a jump, ensuring realistic physics-based movement.

These settings must be carefully balanced to maintain fluidity in movement while ensuring gravity behaves naturally within the game world. If the jumpForce is too high or gravity is too low, movement may feel unnatural and floaty, whereas too much gravity may make jumping feel sluggish or restrictive.

Crouch Parameters

Crouching is often used for stealth mechanics, accessing tight spaces, or avoiding enemy detection. The following parameters allow precise control over how crouching is implemented in the game:

- crouchHeight: Determines the reduced height of the character while crouching, defining how low they can go.
- standingHeight: Specifies the standard height of the character when standing upright.
- timeToCrouch: Controls the duration of the crouch-to-stand and stand-to-crouch transitions, ensuring smooth animations.
- crouchingCenter: Adjusts the character’s center of mass while crouching, preventing collisions or unnatural movement shifts.
- standingCenter: Defines the center position when the character is standing, maintaining balance in movement physics.
- isCrouching: A Boolean flag that checks whether the character is currently crouching, used to trigger gameplay-specific events.
- duringCrouchAnimation: A flag indicating whether the crouch animation is currently in progress, preventing conflicting actions during transitions.

These parameters ensure a smooth crouching system, allowing for seamless interactions with the environment. Properly configured crouching mechanics help improve gameplay flow, particularly in stealth-based games or first-person shooters where taking cover is crucial. For instance, in a **fast-paced** action game, higher sprint speeds and responsive jump mechanics may be prioritized, whereas in a realistic survival game, movement may be slower, with limited sprinting and more grounded jumping physics. By carefully adjusting these parameters, developers can craft a character movement system that feels both intuitive and immersive, ensuring a smooth player experience throughout the game.

The code in Figure 9 contains parameters designed to control the head movement (HeadBob), Interactable and Object Holding features of a first-person perspective player character.

```
[Header("HeadBob Parameters")]
1 reference
[SerializeField] private float walkBobSpeed = 14f;
1 reference
[SerializeField] private float walkBobAmount = 0.05f;
1 reference
[SerializeField] private float sprintBobSpeed = 18f;
1 reference
[SerializeField] private float sprintBobAmount = 0.11f;
1 reference
[SerializeField] private float crouchBobSpeed = 8f;
1 reference
[SerializeField] private float crouchBobAmount = 0.025f;
2 references
private float timer;
5 references
private Vector3 hitPointNormal;
2 references
private float defaultYpos;

[Header("Interactable Parameters")]
1 reference
[SerializeField] private float interactDistance = 2f;
4 references
private bool isInteractable = true;
5 references
private bool isObjectHeld = false;
4 references
private Transform heldObject;
1 reference
private float holdingDistance = 2f;
3 references
private float dropObjectTimer = 0f;
```

Figure 9. Head Movement and Interaction Parameters

A realistic first-person character movement system not only requires smooth motion mechanics but also subtle visual effects to enhance immersion. One such effect is head bobbing, which simulates the natural movement of a person's head while walking, running, or crouching. Additionally, interaction mechanics allow players to engage with objects in the game world, making exploration and gameplay more dynamic. The following parameters help refine both head movement realism and object interaction mechanics, ensuring a more natural and engaging player experience.

HeadBob Parameters

Head bobbing is a subtle but crucial detail that enhances the realism of first-person movement by making the player feel physically present in the game world. These parameters control the frequency and intensity of head movement in different states:

- walkBobSpeed: Determines how fast the head bobs while walking, simulating the natural rhythm of human movement.
- walkBobAmount: Controls the intensity of the head bob effect while walking. A higher value results in more noticeable head movement.
- sprintBobSpeed: Defines the speed of head bobbing while running, as running naturally produces faster and more pronounced movements.
- sprintBobAmount: Adjusts the intensity of the head bob effect when sprinting, ensuring that the visual feedback matches the increased movement speed.
- crouchBobSpeed: Sets the rate at which the head bobs when crouching, usually slower than walking or sprinting.
- crouchBobAmount: Controls how much the head moves when crouching, typically kept subtle to prevent excessive motion in confined spaces.
- Timer: Tracks the time progression of the head bob animation, ensuring smooth and consistent movement cycles.
- hitPointNormal: Stores the normal vector of the surface that the player's head touches, useful for detecting collisions with objects in first-person view.
- defaultYpos: Defines the default vertical position of the camera (head), ensuring the head bob effect returns to a neutral state when standing still.

Together, these parameters create a visually dynamic movement experience. Without head bobbing, movement can feel rigid and unnatural, while excessive head movement can cause discomfort or motion sickness. Properly balanced settings ensure that players feel immersed without experiencing disorientation.

Interactable Parameters

Interactivity is another key aspect of immersive first-person gameplay. The ability to pick up, move, and interact with objects adds depth to the player's engagement with the environment. The following parameters define how the player interacts with objects:

- interactDistance: Determines how far the player can reach to interact with objects. This ensures that objects are neither too difficult to reach nor unrealistically accessible from a distance.
- isInteractable: A Boolean flag that checks whether an object is interactable. This allows the game to dynamically enable or disable interaction prompts based on the player’s position and the nature of the object.

- isObjectHeld: Indicates whether the player is currently holding an object, preventing multiple objects from being grabbed simultaneously.
- heldObject: Stores a reference to the object the player is holding, allowing it to be manipulated dynamically.
- holdingDistance: Defines how far the held object should be positioned from the player's viewpoint. This ensures that objects appear within reach while preventing clipping or unnatural placement.
- dropObjectTimer: Sets the delay before an object is dropped, allowing for controlled object release timing to avoid unintended interactions.

By carefully fine-tuning these parameters, developers can create intuitive and responsive interactions, enhancing player agency within the game world. For instance, in a survival game, a realistic holding distance and weight simulation could be introduced, while in a puzzle game, precise object placement mechanics might be more important. By integrating these mechanics thoughtfully, developers can create a game where movement feels lifelike and interactions feel intuitive, leading to a more engaging and immersive gaming experience.

Figure 10 shows an Interactable class that controls interactable objects and provides the ability to interact with them.

```
using UnityEngine;

4 references
public class Interactable : MonoBehaviour
{
    1 reference
    [SerializeField] private KeyCode interactKey = KeyCode.E;
    2 references
    [SerializeField] private float interactDistance = 2f;

    4 references
    private bool isInteractable = true;
    5 references
    private Transform heldObject;
    7 references
    private Camera playerCamera;

    0 references
    private void Awake()
    {
        playerCamera = Camera.main;

        if (playerCamera == null)
        {
            Debug.LogError("Ana kamera (Main Camera) bulunamad�. Lütfen kodun baş�nda playerCamera değişkenini doğru bir şekilde ayarlad���n�zdan emin olun.");
        }
    }

    0 references
    private void Update()
    {
        HandleInput();
        MoveHeldObjectToCamera();
    }

    1 reference
    private void HandleInput()
    {
        if (Input.GetKeyDown(interactKey))
        {
            if (isInteractable)
            {
                TryInteract();
            }
            else
            {
                DropObject();
            }
        }
    }
```

Figure 10. Interactable class that controls interactable objects

In many games, object interaction mechanics play a crucial role in creating an immersive and engaging experience. Whether it's picking up objects, opening doors, or triggering in-game events, a well-implemented interaction system allows players to engage with their environment intuitively. This class is designed to enable players to interact with objects within a specified distance and manipulate them dynamically, ensuring smooth and realistic gameplay.

Interaction Key and Distance

To initiate an interaction, the script defines specific controls and range limitations:

- interactKey: Determines which key the player must press to interact with objects. By default, this is set to the "E" key, a widely used interaction key in first-person games.
- interactDistance: Specifies the maximum distance within which an object can be interacted with. This ensures that the player cannot interact with objects that are too far away, maintaining a realistic sense of reach and interaction.

By defining these parameters, the system ensures that interactions feel responsive yet grounded in realism, preventing players from interacting with objects in an unnatural or unfair way.

Interactable State and Object Handling

The script also tracks whether an object is currently interactable or being held by the player:

- isInteractable: A Boolean flag indicating whether the player is able to interact with an object. This helps determine if an object is within range and eligible for interaction.
- heldObject: If the player is holding an object, this variable stores a reference to that object, allowing the script to manipulate it (e.g., move it, rotate it, or release it).

By managing these states dynamically, the interaction system ensures that objects behave logically and predictably—for example, preventing the player from picking up multiple objects at the same time or dropping an object unexpectedly.

Player Camera and Perspective

Since this is a first-person interaction system, the script relies on the playerCamera to determine the player’s viewpoint:

- playerCamera: Represents the camera from the player's perspective, ensuring that interactions happen relative to where the player is looking.

This ensures that when the player presses the interaction key, the system checks the object directly in their line of sight, making interactions feel natural and intuitive.

Initialization with the Awake Method

When the game starts, the Awake() method is triggered automatically:

- Awake(): This method is called as soon as the class is created. It ensures that the Main Camera is properly set, preventing errors related to missing camera references.

By performing this setup early, the script ensures that all interaction logic functions correctly right from the beginning of the game.

Real-Time Input Processing in the Update Method

To make interactions feel immediate and responsive, the script listens for player input every frame:

- Update(): Called every frame, this method continuously listens for user input and checks if an interaction should be triggered.

Since Update() runs in real-time, the player can seamlessly interact with objects without delays or unresponsiveness, ensuring a fluid gameplay experience.

Handling Input with the HandleInput Method

The HandleInput() method is responsible for processing the player's interaction attempts:

- If the interaction key is pressed and an object is available for interaction, the method initiates the interaction.
- If the player is already holding an object, pressing the interaction key again releases the object, preventing conflicts between interactions.

This method ensures that interactions remain consistent and predictable, avoiding situations where the player unintentionally picks up or drops objects. Whether used for picking up objects, activating doors, or triggering in-game events, this system provides a solid foundation for interactive gameplay, making the game world feel more alive and engaging.

Figure 11 contains a method called TryInteract(). This method attempts to interact with the objects around the player when he presses the interact button.

```csharp
1 reference
private void TryInteract()
{
    if (playerCamera == null)
    {
        Debug.LogError("Player kamera referansı null. Kodunuzun başında playerCamera değişkenini doğru şekilde ayarladığınızdan emin olun.");
        return;
    }

    RaycastHit hit;

    if (Physics.Raycast(playerCamera.transform.position, playerCamera.transform.forward, out hit, interactDistance))
    {
        if (hit.collider != null)
        {
            Interactable interactable = hit.collider.GetComponent<Interactable>();

            if (interactable != null && interactable.CanInteract())
            {
                DropObject();
                heldObject = interactable.transform;
                isInteractable = false;
            }
            else
            {
                Debug.LogWarning("Interactable bileşeni eksik, CanInteract() false döndü veya başka bir nedenle null. Nesne adı: " + hit.collider.gameObject.name);
            }
        }
        else
        {
            Debug.LogWarning("Raycast sonucunda bir collider bulunamadı. Nesne adı: " + hit.collider.gameObject.name);
        }
    }
    else
    {
        Debug.LogWarning("Raycast başarısız oldu. Muhtemelen hiçbir nesne algılanmadı.");
    }
}
```

Figure 11. TryInteract() method

In first-person games, interactions between the player and the game world are an essential part of immersive gameplay. Whether it's picking up objects, pressing buttons, or activating mechanisms, an interaction system must accurately detect objects within range and process interactions dynamically.

This method is responsible for determining which objects the player can interact with, checking various conditions, and initiating interaction logic when applicable. The process follows a structured sequence, ensuring smooth and logical interaction mechanics.

Player Camera Control

Before executing any interaction logic, the method first ensures that the player's camera is correctly referenced:

- playerCamera == null: If the player camera reference is missing (null), the script prints an error message and prevents further execution.
- This check prevents potential null reference errors, ensuring that the interaction system only runs when the required camera component is properly set up.

By verifying the camera's existence, the method guarantees that all interactions occur from the correct viewpoint, ensuring that objects are detected accurately based on the player's perspective.

Raycast Process (Detecting Objects in Front of the Player)

Once the camera reference is validated, the method uses Raycasting to determine which objects are within interaction range:

- Physics.Raycast(playerCamera.transform.position, playerCamera.transform.forward, out hit, interactDistance):
  - A Raycast is projected from the player's viewpoint (camera position) in the direction the player is looking (camera forward).
  - This ray extends up to the specified interactDistance and detects any objects in its path.
  - If an object is hit, its information is stored in the hit variable, allowing the system to process interactions based on the detected object.

Raycasting is a crucial technique in game development, as it simulates the player’s line of sight, ensuring that interactions only occur with objects directly in front of the player, rather than those behind or outside their view.

Collider Control (Ensuring the Object is Valid for Interaction)

Once the Raycast detects an object, the next step is to verify that the object has a Collider component:

- hit.collider != null:
  - This condition checks whether the detected object actually has a collider.
  - If the object lacks a collider, interaction is not processed, as colliders define the physical boundaries of objects and enable proper detection.

Ensuring that an object has a valid collider prevents interaction errors and ensures that only solid, interactable objects are detected while ignoring empty space.

Checking for an Interactable Object

Once the collider is confirmed, the script then determines whether the detected object is interactable:

- Interactable interactable = hit.collider.GetComponent<Interactable>();
  - This checks whether the object has the Interactable component attached to it.
  - If the object does not have an Interactable script, it is ignored, as it is not intended for interaction.

By requiring objects to have a specific Interactable component, the system ensures that only designated objects can be interacted with, avoiding unintended behaviors with non-interactable objects.

Interaction Status Check (Can the Object be Interacted With?)

Once the object is confirmed as interactable, the method then checks whether interaction is currently allowed:

- interactable != null && interactable.CanInteract():
  - The method calls CanInteract() on the detected object.
  - If CanInteract() returns true, the object is ready to be interacted with, and the system proceeds with the interaction logic.

This additional check ensures that certain objects can have dynamic interaction states—for example, an object might only be interactable under specific conditions, such as after completing a quest or unlocking a door.

Object Movement and State Changes (Handling Interaction Logic)

Once all the conditions for interaction are met, the script then handles object movement and interaction state updates:

- DropObject():
  - If the player is already holding an object, they must drop it first before interacting with a new one.
  - This ensures that only one object can be held at a time, preventing bugs such as carrying multiple objects simultaneously.
- heldObject = interactable.transform:
  - If the player is not already holding an object, the script assigns the interacted object to the heldObject variable, effectively "picking it up."
  - This allows the object to be moved dynamically until the player releases it.
- isInteractable = false:
  - Once an object is picked up, its interactable state is disabled, ensuring that it cannot be interacted with again while already being held.
  - This prevents duplicate interactions or unintentional object duplication.

By following these steps, the interaction system ensures smooth and logical transitions between detecting an object, checking interaction conditions, and processing object manipulation.

The code fragment shared in Figure 12 contains the methods that form the continuation of the Interactable class.

```
2 references
public bool CanInteract()
{
    return isInteractable;
}

2 references
private void DropObject()
{
    if (heldObject != null)
    {
        isInteractable = true;
        heldObject = null;
    }
}

1 reference
private void MoveHeldObjectToCamera()
{
    if (heldObject != null)
    {
        Vector3 targetPosition = playerCamera.transform.position + playerCamera.transform.forward * interactDistance;
        heldObject.position = targetPosition;
    }
}
}
```

Figure 12. Methods that form the continuation of the Interactable class.

In first-person games, interacting with objects in a realistic and responsive manner is crucial for immersion. The following methods provide the core functionality for determining whether an object can be interacted with, handling object movement, and ensuring a smooth drop mechanism. These methods work together to create a cohesive and intuitive interaction system, allowing the player to pick up, move, and drop objects seamlessly.

CanInteract() Method – Checking Interaction Availability

Before interacting with an object, the game must determine whether the interaction is currently allowed.

- public bool CanInteract():
    - This method returns a boolean value (true or false) indicating whether the object is interactable.
    - It checks the isInteractable status, which dictates whether the object can be picked up or manipulated.

By using this method, other parts of the script can quickly verify whether an object is available for interaction, preventing unintended interactions and ensuring logical control flow. For example, if an object is already held by another player in a multiplayer game, CanInteract() might return false, preventing duplicate pickups.

DropObject() Method – Handling Object Release

When the player wishes to release a held object, the DropObject() method is triggered.

- private void DropObject():
    - Before executing the drop action, the method checks if heldObject is not null, ensuring that the player is actually holding something.

- If a valid object is being carried:
  - isInteractable is set to true, allowing the object to be interacted with again.
  - heldObject is set to null, effectively "dropping" the object.

This method ensures that objects are released smoothly and return to an interactable state, allowing for fluid and intuitive gameplay interactions.

Without this mechanism, objects could remain stuck in an "uninteractable" state after being picked up, making them inaccessible for future interactions.

MoveHeldObjectToCamera() Method – Moving Objects in Front of the Player

Once an object is picked up, it must be positioned properly in front of the player to ensure a natural holding motion.

- private void MoveHeldObjectToCamera():
  - If an object is being held, this method moves it a specific distance in front of the player's camera.
  - The new targetPosition is calculated by adding a set distance to the player's camera position and forward direction, ensuring that the object is always positioned in front of the player's viewpoint.
  - The heldObject is then moved smoothly to this target position, preventing abrupt or unnatural repositioning.

This method ensures that the held object remains in a consistent and realistic position while allowing natural movement adjustments as the player looks around. For example, if a player picks up a flashlight, MoveHeldObjectToCamera() ensures that the flashlight follows their view direction, allowing them to point it naturally without awkwardly repositioning it manually.

CONCLUSION

Today, environmental issues have become a serious concern worldwide. In this paper, a 3D game was developed using design-based research methodology to raise awareness about garbage sorting and recycling issues in an educational game. The game aims not only to provide a fun gaming experience but also to raise environmental awareness and encourage participation in the recycling movement. In addition to these, it is aimed to have fun by having fun and learning at the same time by being inspired by today's style games so that game-loving friends can play in a way that will not bore them. The game is designed to be played by individuals of all ages in order to provide a fun experience and inspire environmental responsibility.

The Unity 3D game engine used in the game contributed to the successful realization of the project with its user-friendly interface, rapid prototyping capabilities, flexibility in 2D and 3D game development, physics and animation systems, and support for various platforms. Unity 3D's rich ecosystem of communities and resources made the development process more efficient.

The map encourages the proper use of garbage bins and recycling in the city. The use of separate bins for different types of garbage, the placement of informative signs and the realistic modeling of the streets reinforce the environmental message of the game.

This study also describes in detail the important pieces of code used in the game. The codes of features such as character control, head movement, interaction and object handling provide the basic functionality of the game. It is expected that the codes shared will contribute to the development of games using the Unity 3D engine or to guide simulation environment developers in interdisciplinary fields.

The game aims to provide players with a fun experience while raising awareness of environmental responsibility. As a result, this educational game is thought to be an important step in drawing attention to environmental problems and raising awareness of individuals about garbage sorting and recycling. It is emphasized that each player can become an environmentally friendly individual by making small changes.

Such initiatives to leave a clean and sustainable world for future generations can increase the general environmental awareness of the society and contribute to positive changes.

In this game, players can learn many different skills, giving them the opportunity to develop environmental awareness. Here are some of the skills this game teaches:

- Recycling and sustainability awareness: Players need to recycle garbage by using recycling bins correctly. This can make players more aware of sustainability and waste management.
- Observation and analysis skills: Players need to recognize the types of garbage before they can put it in the correct recycling bin. This can improve players' observation and analytical skills.

RECOMMENDATIONS

Educational institutions should focus more on environmental awareness education and use new instructional technologies, such as design-based exploratory games, to impart environmentally friendly values to students. Parents should guide children's choice of mobile games towards

educational and environmental awareness-oriented games. Game selection should be appropriate for children's development and game durations should be limited.

Game designers should develop UNITY-based educational games taking into account the age and developmental level of children and players. The games should have colorful and expressive graphics to attract children's attention and sustain their interest. Furthermore, games should have simple user interface and clear instructions so that children can easily use and understand the game. Games should include tasks, puzzles and activities that promote environmental awareness. For example, players can learn about the importance of recycling, learn about saving energy, or perform tasks to protect wildlife.

In this way, players learn important concepts about the environment while having fun. Games can have multiplayer features to encourage children's social interaction. By playing with friends, children can learn to cooperate, develop communication skills and teamwork. Games should include a progression system that tracks and rewards children's achievements. For example, children can earn rewards for completing certain tasks or achieving certain points. This increases children's motivation and makes the game more engaging. Parents and teachers should keep track of children's play experiences and engage in conversations about games to contribute to environmental awareness education. It is important to encourage and support children by emphasizing the learning opportunities that games provide.

REFERENCES

Barab, S., & Squire, K. (2016). Design-based research: Putting a stake in the ground. *In Design-based Research (pp. 1-14).* Psychology Press.

Blackman, S. (2013). Beginning 3D Game Development with Unity 4: All-in-one, multi-platform game development. Apress. https://doi.org/10.1007/978-1-4302-4900-9

Bond, J. G. (2014). Introduction to Game Design, Prototyping, and Development: From Concept to Playable Game with Unity and C. addison-wesley professional.

Bressler, D. M., & Bodzin, A. M. (2013). A mixed methods assessment of students' flow experiences during a mobile augmented reality science game. *Journal of computer assisted learning, 29*(6), 505-517. https://doi.org/10.1111/jcal.12008

Connolly, T. M., Boyle, E. A., MacArthur, E., Hainey, T., & Boyle, J. M. (2012). A systematic literature review of empirical evidence on computer games and serious games. *Computers & education, 59*(2), 661-686. https://doi.org/10.1016/j.compedu.2012.03.004

De Freitas, S. (2006). *Learning in immersive worlds: A review of game-based learning.* http://researchrepository.murdoch.edu.au/id/eprint/35774/1/gamingreport_v3.pdf.

Dondlinger, M. J. (2007). Educational video game design: A review of the literature. *Journal of applied educational technology, 4*(1), 21-31.

Ebner, M., & Holzinger, A. (2007). Successful implementation of user-centered game based learning in higher education: An example from civil engineering. Computers & education, 49(3), 873-890. https://doi.org/10.1016/j.compedu.2005.11.026

Esen, A., & fevzi Esen, M. (2018). Çevre Eğitimi ve Bilinci Araştırması. *Akademik Bakış Uluslararası Hakemli Sosyal Bilimler Dergisi*, (65), 164-178. https://dergipark.org.tr/tr/pub/tabin/issue/75873/12 51424

Gee, J. P. (2003). What video games have to teach us about learning and literacy. *Computers in entertainment (CIE), 1*(1), 20-20. https://doi.org/10.1145/950566.950595

Güneş, M. (2021). *El Yapımı Patlayıcılara Müdahalede Sanal Gerçeklik ile Kazaların Önlenmesine Yönelik Bir Uygulama* (Doctoral dissertation, Doktora Tezi, Gazi Üniversitesi Fen Bilimleri Enstitüsü, Ankara).

Hainey, T., Connolly, T. M., Boyle, E. A., Wilson, A., & Razak, A. (2016). A systematic literature review of games-based learning empirical evidence in primary education. *Computers & Education, 102,* 202-223. https://doi.org/10.1016/j.compedu.2016.09.001

Huizenga, J., Admiraal, W., Akkerman, S., & Dam, G. T. (2009). Mobile game-based learning in secondary education: engagement, motivation and learning in a mobile city game. *Journal of computer assisted learning, 25*(4), 332-344. https://doi.org/10.1111/j.1365-2729.2009.00316.x

Jackson, S. (2015). *Unity 3D UI essentials*. Packt Publishing.

Jangraw, D. C., Johri, A., Gribetz, M., & Sajda, P. (2014). NEDE: An open-source scripting suite for developing experiments in 3D virtual environments. *Journal of neuroscience methods*, 235, 245-251. https://doi.org/10.1016/j.jneumeth.2014.06.033

Kiziroğlu, İ. (2023). Çevre eğitimi ve çevre bilinci. *Tabiat ve İnsan, 2*(193), 5-17. https://dergipark.org.tr/tr/pub/tabin/issue/75873/12 51424

Lavieri, E. (2018). *Getting Started with Unity 2018: A Beginner's Guide to 2D and 3D game development with Unity*. Packt Publishing Ltd.

Liao, C. C., Chen, Z. H., Cheng, H. N., Chen, F. C., & Chan, T. W. (2011). My-Mini-Pet: A handheld pet-nurturing game to engage students in arithmetic practices. *Journal of*

Computer Assisted Learning, 27(1), 76-89. https://doi.org/10.1111/j.1365-2729.2010.00367.x

Malone, T. W. (1980, September). *What makes things fun to learn? Heuristics for designing instructional computer games.* In Proceedings of the 3rd ACM SIGSMALL symposium and the first SIGPC symposium on Small systems (pp. 162-169).

Malone, T. W., & Lepper, M. R. (2021). *Making learning fun: A taxonomy of intrinsic motivations for learning*. In Aptitude, learning, and instruction (pp. 223-254). Routledge.

McKenney, S., & Reeves, T. (2018). Conducting educational design research. Routledge. https://doi.org/10.4324/9781315105642

Oblinger, D. G. (2004). The next generation of educational engagement. Journal of interactive media in education, 2004(1), 10-10. https://doi.org/10.5334/2004-8-oblinger

Okita, A. (2019). *Learning C# programming with Unity 3D.* AK Peters/CRC Press. https://doi.org/10.1201/9780429810251

Papastergiou, M. (2009). Digital game-based learning in high school computer science education: Impact on educational effectiveness and student motivation. *Computers & education, 52*(1), 1-12. https://doi.org/10.1016/j.compedu.2008.06.004

Sánchez, J., & Olivares, R. (2011). Problem solving and collaboration using mobile serious games. *Computers & Education, 57*(3), 1943-1952. https://doi.org/10.1016/j.compedu.2011.04.012

Schwabe, G., & Göth, C. (2005). Mobile learning with a mobile game: design and motivational effects. *Journal of computer assisted learning, 21*(3), 204-216. https://doi.org/10.1111/j.1365-2729.2005.00128.x

Sourmelis, T., Ioannou, A., & Zaphiris, P. (2017). Massively Multiplayer Online Role Playing Games (MMORPGs) and the 21st century skills: A comprehensive research review from 2010 to 2016. *Computers in Human Behavior, 67*, 41-48. https://doi.org/10.1016/j.chb.2016.10.020

Tuğtekin, U., & Kaleci, D. (2011). *3D modelleme tekniği kullanılarak bilgisayar oyunu tasarımı*. XIII. Akademik Bilişim, Malatya.

Tüzün, H., Yılmaz-Soylu, M., Karakuş, T., Inal, Y., & Kızılkaya, G. (2009). The effects of computer games on primary school students' achievement and motivation in geography learning. *Computers & education, 52*(1), 68-77. https://doi.org/10.1016/j.compedu.2008.06.008

Tykoski, S. (2022). *Mastering Game Design with Unity 2021: Immersive Workflows, Visual Scripting, Physics Engine, GameObjects, Player Progression, Publishing, and a Lot More (English Edition).* BPB Publications.

Unity Technologies 2005. *Unity- Manual: About Unity*. http://docs.unity3d.com/Manual/UnityOverview.html

Watson, M. R., Voloh, B., Thomas, C., Hasan, A., & Womelsdorf, T. (2019). USE: An integrative suite for temporally-precise psychophysical experiments in virtual environments for human, nonhuman, and artificially intelligent agents. *Journal of neuroscience methods*, 326, 108374. https://doi.org/10.1016/j.jneumeth.2019.108374

Wong, L. H., Hsu, C. K., Sun, J., & Boticki, I. (2013). How flexible grouping affects the collaborative patterns in a mobile-assisted Chinese character learning game? *Journal of Educational Technology & Society, 16*(2), 174-187.