Design and implementation of modular animated agents in VR using inverse kinematics and XR Toolkit

Oleksiy Sinkevych[1,*,†], Yuliia Sapsa[2,†] and Oleksandr Storozhuk[1,†]

[1]Ukrainian National Forestry University, 103 Henerala Chuprynky Str., Lviv, 79057, Ukraine
[2]University of Valladolid, Pl. Campus Universitario, 1, 47011 Valladolid, Spain

Abstract

This paper presents a reproducible pipeline for building interactive animated characters in virtual reality (VR) environments, aimed at educational and small-team research settings. The pipeline couples Blender for skeletal rigging and keyframe animation with Unity's XR Interaction Toolkit for runtime control and interaction handling. We formalize the VR scene as a set of agents, interactive objects, and spatial trigger zones; drive character behavior with a finite-state machine over baked animation clips; and route user input, collisions, and physics-based responses through XR controllers. A functional prototype – a low-polygon VR shooter – demonstrates the feasibility of the pipeline end to end and serves as a testbed for animation control and interaction logic. Its modular, component-based architecture can be extended toward training simulators and other interaction-intensive VR scenarios, and provides a starting point for work on human–agent interaction, behavioral modeling, and procedural content generation.

Keywords

3D modeling, rigging, Blender, Unity engine, C#, Game development, Human-computer interaction

1. Introduction

Virtual reality (VR) has become an increasingly relevant field of research due to its growing application in simulation, training, and digital interaction design [1]. The development of realistic and responsive 3D characters within immersive VR environments is a crucial component in ensuring both functional and emotional engagement of the user [2]. Despite the availability of mature game engines and modeling tools, the process of integrating animated characters into VR applications remains insufficiently formalized, particularly for educational use cases and low-cost development settings [3].

This paper addresses that gap by synthesizing a coherent pipeline for creating and controlling interactive 3D characters in VR from open-source modeling tools and a freely available game engine. We focus on the design, animation, and integration of low-polygon humanoid characters, using Blender for skeletal rigging and animation and Unity with the XR Interaction Toolkit for real-time deployment and user interaction [4, 5, 6]. To exercise the pipeline end to end, we developed a prototype VR system in the form of a low-polygon mini-shooter, which serves as a working demonstration of animation control, interaction mechanics, and system responsiveness.

The methodology is grounded in a formal decomposition of the virtual environment into agents, interactive objects, and spatial interaction zones [7]. The animation subsystem is based on a skeletal rig augmented with inverse kinematics (IK) at authoring time, which allows humanoid motion to be posed and keyframed efficiently [8, 9]. Behavioral logic is implemented as a small set of C# scripts that define character states and transitions together with event-driven responses to user input and collisions.

CS&SE@SW 2025: 8th Workshop for Young Scientists in Computer Science & Software Engineering, November 21, 2025, Kryvyi Rih, Ukraine
https://doi.org/10.55056/cssesw2025/paper20
*Corresponding author.
†These authors contributed equally.
oleksiy1694@gmail.com (O. Sinkevych); yuliia.sapsa23@estudiantes.uva.es (Y. Sapsa); storozhuk@nltu.edu.ua (O. Storozhuk)
0000-0001-6651-5494 (O. Sinkevych); 0009-0004-8137-836X (Y. Sapsa); 0000-0001-6566-5271 (O. Storozhuk)

The interaction layer is built on the XR Interaction Toolkit, which abstracts controller input, object manipulation, and animation triggering, backed by physical collision detection.

2. Materials and methods

This section describes the toolchain, the formal model of the virtual scene, and the animation and interaction logic on which the prototype rests [10].

Toolchain and development environment. The pipeline integrates two platforms: Blender 4.3.2 for 3D modeling and animation, and Unity for real-time interaction and deployment. Blender was used to create low-polygon humanoid characters with skeletal rigging, inverse kinematics (IK), and keyframed animation sequences [11]. Character meshes were exported in FBX format with bone hierarchies and keyframe data preserved [12]. Unity was chosen for its support of VR development and its integration with the XR Interaction Toolkit [13], which provides APIs for controller-based interaction, object grabbing, trigger events, and locomotion.

All scripting was done in C#, using Visual Studio Code with the Unity development plugins. Testing and debugging relied on the XR Device Simulator, which allowed interaction to be exercised on a desktop machine without continuous use of a headset.

Formal model of the virtual scene. The virtual environment is modeled as a dynamic system S, consisting of several interacting components:

$$S = \{P, E, O, Z\} \tag{1}$$

where: P is the player agent, modeled as a VR rig with head and hand tracking;
$E = \{e_1, e_2, \dots, e_n\}$ is the set of enemy agents (non-player characters, NPCs);
$O = \{o_1, o_2, \dots, o_m\}$ is the set of interactable objects (e.g., weapons);
$Z = \{z_1, z_2, \dots, z_k\}$ is the set of spatial trigger zones, defined by colliders.

Each object in the scene has an associated state space Q_i, and interactions are defined as state transitions triggered by discrete events.

Animation framework and rigging model. The enemy character was modeled as a simplified low-polygon mesh with humanoid proportions. A full-body skeleton was generated with Blender's Rigify add-on: a human meta-rig was fitted to the mesh and then used to generate a control rig whose limb chains carry inverse-kinematics constraints. Bone weights were assigned with Blender's automatic skinning algorithm and refined by hand at the critical joints (shoulders, elbows, knees).

IK enters the pipeline only at authoring time. The IK chains belong to the control rig in Blender, where they let the animator pose a limb by moving a foot or hand instead of rotating each joint in turn; the clips exported to Unity are baked keyframe data, and no IK is solved at runtime.

The following animation clips were produced:

- Idle (neutral pose);
- Alert (reaction to player presence);
- Dead (triggered by projectile collision).

The clips were authored at the scene frame rate of 24 fps and kept short: the alert clip, for instance, spans 35 frames, or roughly 1.5 s. Transitions between them are driven by Unity's Mecanim Animator Controller [14] under a finite-state machine:

$$M = (Q, \Sigma, \delta, q_0, F) \tag{2}$$

where: $Q = \{Idle, Alert, Dead\}$ is the set of states;
$\Sigma = \{TriggerEnter, Collision\}$ is the event set;
$\delta : Q \times \Sigma \to Q$ is the transition function;
$q_0 = Idle$ is the initial state and $F = \{Dead\}$ the final state.

TriggerEnter carries the agent from *Idle* to *Alert* when the player enters its activation zone; *Collision* carries it from either state to *Dead*. Both events map onto Animator triggers (`Activate` and `Death`) set from C#, so the transitions are deterministic.

Interaction logic and control system. Interaction with the VR scene is handled by Unity's XR Interaction Toolkit, which separates the roles of Interactors (the controllers) and Interactables (weapons, enemies). Interaction events are dispatched through Unity's event system [15], and scripts respond to them in collision and trigger callbacks such as `OnTriggerEnter` and `OnCollisionEnter`.

Projectile interaction follows a physics-driven model [16]. When the user presses the shoot button, a rigid-body projectile is instantiated with a directional velocity:

$$v_{bullet} = d \cdot v_0, \tag{3}$$

$$d = \text{forward}(C_{controller}), \tag{4}$$

where: v_{bullet} is the initial velocity vector of the projectile, v_0 is the predefined scalar speed, and $C_{controller}$ is the controller transform from which the forward direction d is obtained.

A hit is registered when the projectile's collider intersects that of an agent and the agent carries the expected tag:

$$C_{bullet} \cap C_{enemy} \neq \emptyset \ \text{ and } \ \text{tag}(enemy) = \texttt{Enemy}. \tag{5}$$

When both conditions hold, the agent transitions to the *Dead* state and the corresponding death animation is triggered.

System architecture. The system follows a component-based architecture in which each GameObject carries one or more behavior scripts [17]:

- `Fire.cs` – instantiates projectiles;
- `TargetDummy.cs` – handles state transitions and animation triggers for agents;
- `EnemyCharacterTrigger.cs` – manages activation zones;
- `HandsAnimation.cs` – drives hand-gesture animation from controller input.

Control flow combines Unity's update loop with event-driven callbacks. Each script operates independently and subscribes to shared events only where it needs to, so components can be replaced or reused without touching the others.

3. Results

The implemented system is a compact virtual reality application, "Low Poly Strike", built as a functional prototype of the animation–interaction pipeline described above. It is intended as a working demonstration and as a testbed for later studies of VR character behavior, human–agent interaction, and immersive game mechanics, rather than as a finished game.

Prototype and virtual environment. The environment uses a low-polygon aesthetic, which keeps geometric complexity low while retaining the spatial cues [18] that the interaction tasks require, and which lowers the rendering cost on mid-range VR hardware.

The scene contains four elements:

1. the player spawn area;
2. a weapon station holding interactable objects;
3. a trigger zone;
4. an animated humanoid agent acting as a reactive target.

The player is embodied through a rigged XR rig tracked by the headset and controllers. Unity's XR Input Subsystem [19] supplies head and hand poses, so virtual objects can be manipulated through positional tracking and trigger presses. Entering the designated zone activates the enemy agent and moves its animation state from *Idle* to *Alert* – the visible result of coupling event-driven logic to the animation state machine.

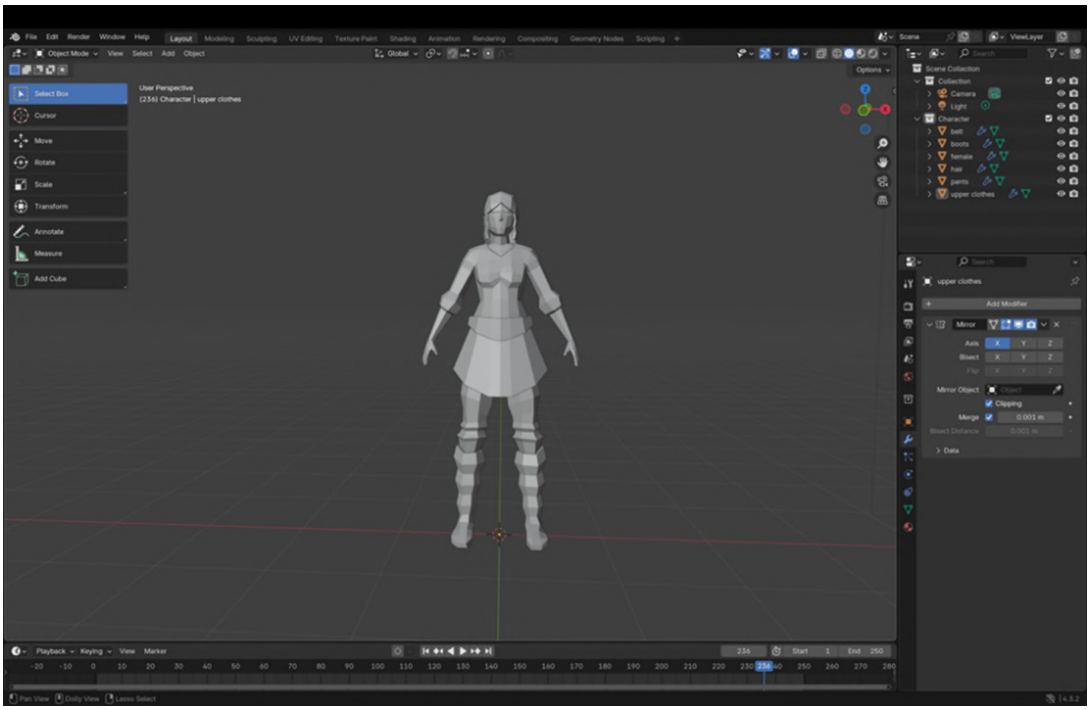

Figure 1: Mirror modifier configured on the *upper clothes* object: mirroring across the X axis, with clipping enabled and a 1 mm merge threshold.

Character modeling. The agent is a custom humanoid modeled in Blender with symmetrical low-poly editing and constructive polygonal operations. Symmetry was maintained with the Mirror modifier [20], which duplicates geometry across a chosen axis and so halves the modeling work while guaranteeing that both sides match. The modifier was applied to the *upper clothes* object forming the character's torso, mirroring across the X axis, with Clipping enabled to keep vertices from crossing the symmetry plane and a merge threshold of 1 mm to fuse the mesh cleanly at the centerline (figure 1). The same configuration was reused for the limbs and the head covering.

Before rigging, face normals were recalculated across the body mesh so that their orientation is consistent (figure 2). Inverted faces are easy to introduce during mirrored and extruded editing, and they produce shading artifacts and unpredictable deformation once the mesh is skinned.

Rigging and animation. A human meta-rig was added to the scene and scaled to the character's proportions, after which each bone was aligned in Edit Mode with the body part it drives (figure 3). Accurate placement at this stage is what makes the deformation anatomically plausible later, since the bone positions determine how the surrounding geometry follows the skeleton.

The mesh was then bound to the armature with the Armature Deform with Automatic Weights operation [21], which creates vertex groups assigning each vertex to one or more bones according to spatial proximity and mesh topology; these weights define how the surface deforms when the bones are transformed. Generating the control rig from the meta-rig produced the animator-facing layer shown in figure 4: custom bone shapes, bone collections separating face and body controls, and IK chains on the arms and legs that allow a limb to be posed from its end effector, so that bending a knee or an elbow requires moving only the foot or the hand [22]. Rotation limits and transform locks on the joints keep poses within an anatomically plausible range.

Three animation states – idle, alert, and death – were then keyframed in Blender's animation editor and exported as FBX files with the skeleton data embedded. Figure 5 shows the alert clip, held in the project under the name `Attack_preparation`, being edited against the rig. In Unity, an Animator Controller maps gameplay events onto these clips: an `OnCollisionEnter` event sets the `Death` trigger, which plays the death animation and disables further interaction with the object.

Interaction and runtime behavior. Weapons are placed near the spawn zone as interactable prefabs

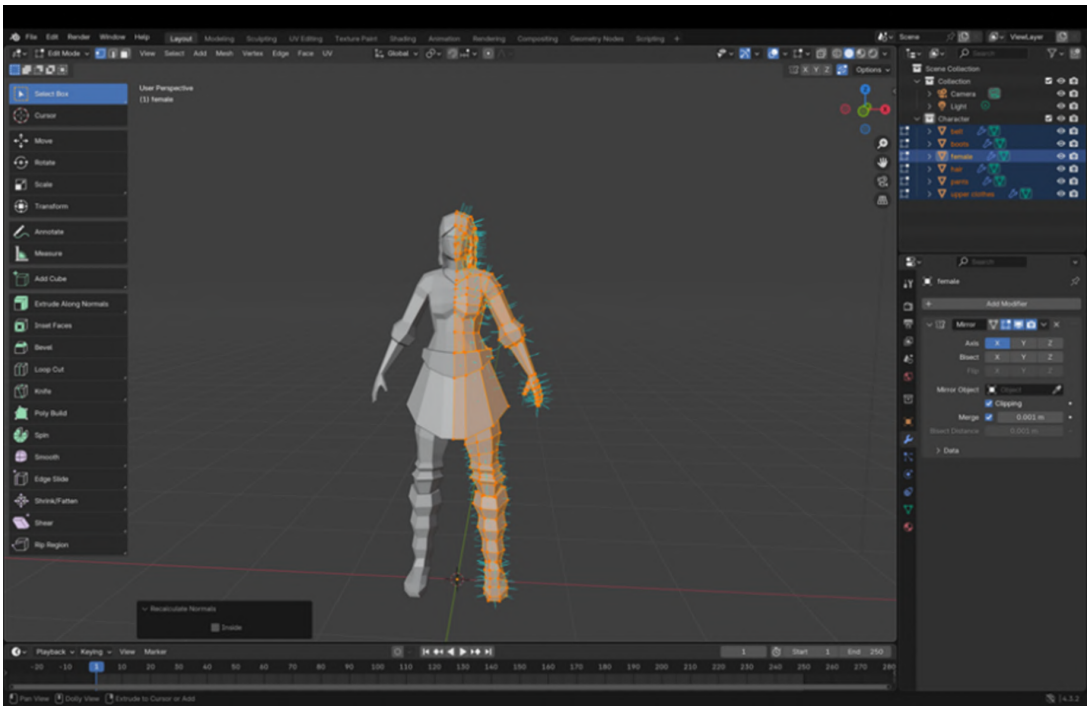

Figure 2: Recalculating face normals on the character mesh prior to rigging.

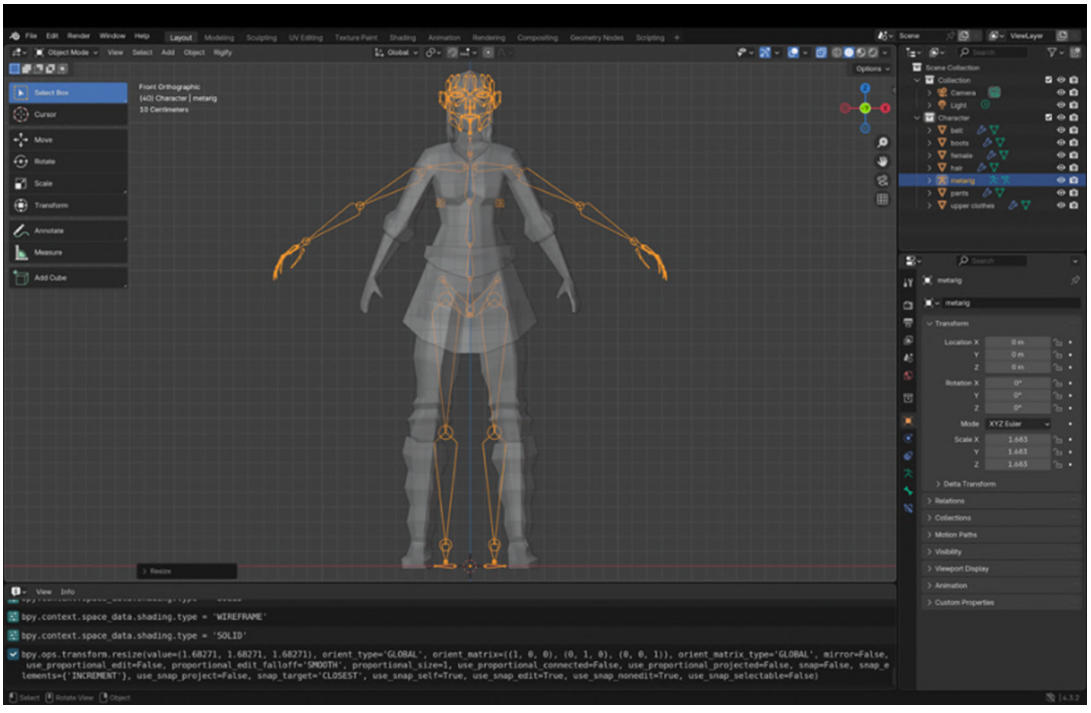

Figure 3: Scaling and positioning skeleton bones according to character body parts.

configured through the XR Interaction Toolkit. Each carries a grab interactor, a physics collider, and a C# behavior that binds the controller's trigger input to projectile instantiation. The projectile is spawned aligned with the orientation of the user's dominant-hand controller and released as a rigid body, so its trajectory follows from Unity's physics rather than from a scripted path.

Figure 6 gives the block diagram of the weapon interaction algorithm: the player picks up a weapon, pulls the trigger, `FireBullet` spawns a projectile, and a collision with the target agent plays the death

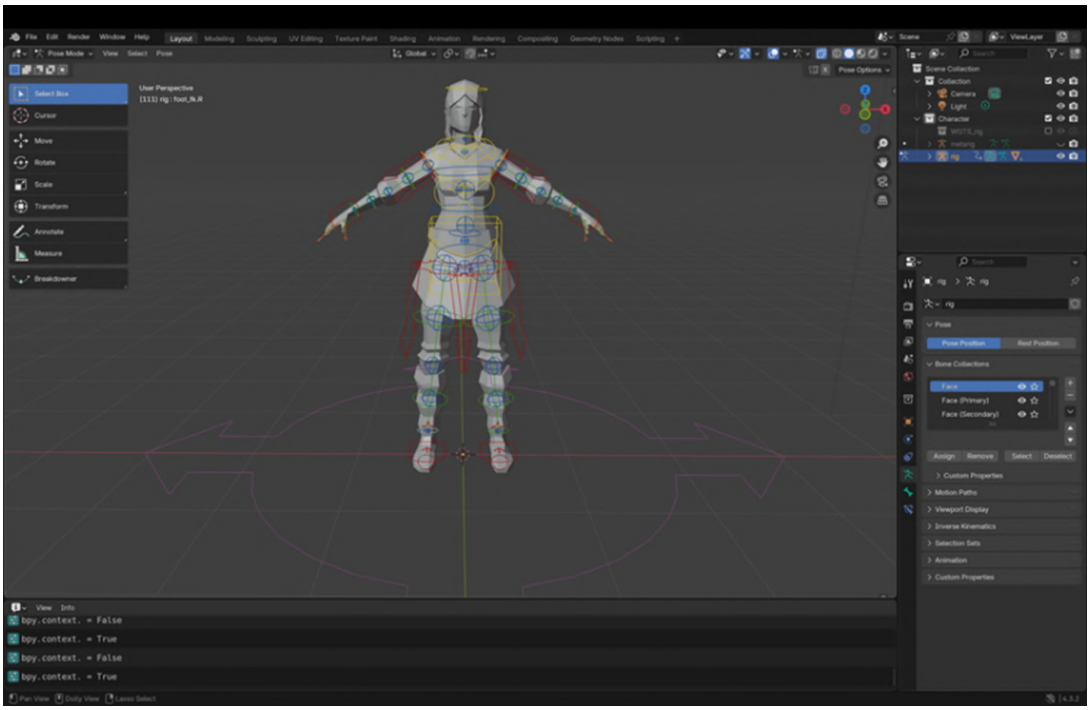

Figure 4: Generated control rig in Pose Mode, with IK-enabled limb chains and separate bone collections for face and body controls.

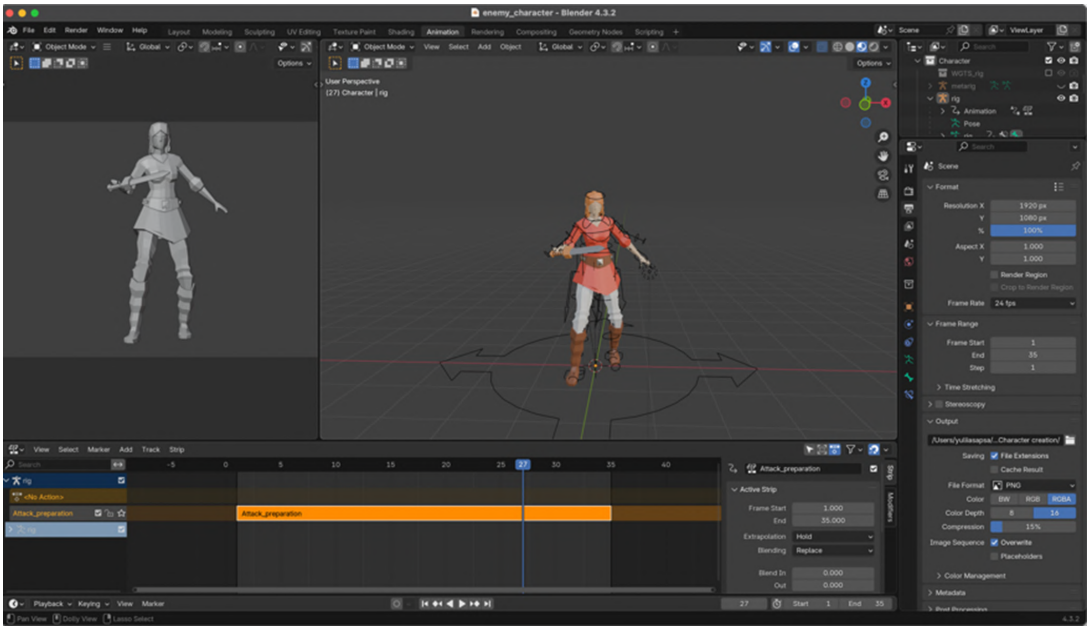

Figure 5: Authoring the alert animation clip (`Attack_preparation`) against the character rig in Blender's animation workspace.

animation and closes the interaction. Encoding the sequence this way keeps the runtime behavior deterministic and makes it straightforward to trace during debugging.

The runtime elements – the agent, the projectile emitter, the trigger zones, and the interaction logic – are all prefabs with serialized parameters [23], so a scenario can be reconfigured from the Unity inspector without editing source code. `OnTriggerEnter` and `OnCollisionEnter` detect the spatial events that drive state changes; together with the update loop they form the event-driven core of the prototype. The full gameplay loop – entering the scene, acquiring the weapon, aiming, firing, and the agent's response – runs end to end.

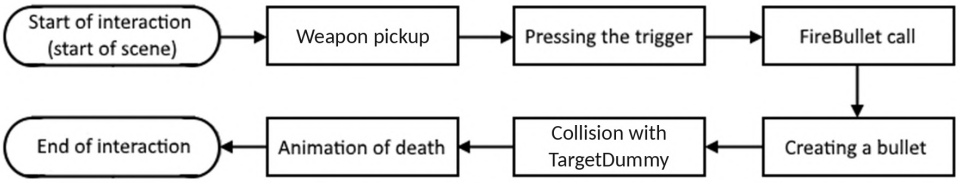

Figure 6: Block diagram of the algorithm for player interaction with weapons.

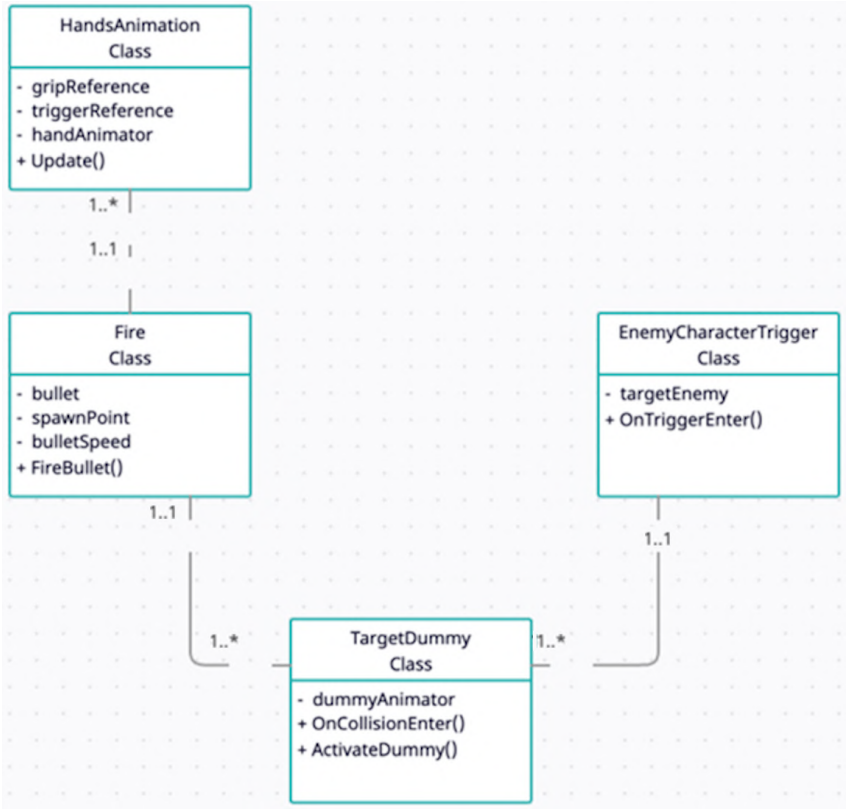

Figure 7: View of a UML class diagram that displays relationships between the main components.

Software architecture. Figure 7 shows the UML class diagram [24] of the four classes that carry the runtime behavior: `HandsAnimation`, `Fire`, `TargetDummy`, and `EnemyCharacterTrigger`.

`HandsAnimation` visualizes the user's hand gestures inside the virtual environment. It reads analog trigger and grip values from the XR controller through `InputActionReference` objects and forwards them to the hand animator, which blends between gesture poses in real time:

```
float gripValue = gripReference.action.ReadValue<float>();
handAnimator.SetFloat("Grip", gripValue);
```

The class does not implement shooting itself, but the gesture state it maintains is the visual counterpart of the input that `Fire` acts on.

`Fire` encapsulates the shooting mechanic. Its method `FireBullet()` instantiates a projectile prefab at a designated `spawnPoint`, gives it an initial velocity along the spawn point's forward axis, and schedules its destruction after five seconds. The prefab carries both a `Rigidbody` and a `Collider`, so it participates in the physics of the scene:

```
public void FireBullet()
{
        GameObject spawnedBullet = Instantiate(
                bullet, spawnPoint.position, spawnPoint.rotation);
        spawnedBullet.GetComponent<Rigidbody>().velocity = spawnPoint.forward * bulletSpeed;
        Destroy(spawnedBullet, 5f);
}
```

A single `Fire` component may strike any number of targets over a session, so it stands in a one-to-many relation to `TargetDummy`.

`TargetDummy` models the behavior of reactive targets. Its `OnCollisionEnter()` method listens for collisions with objects tagged `Weapon` or `Bullet`; on a hit it raises the `Death` trigger and disables the collider, so the agent cannot be hit again:

```
private void OnCollisionEnter(Collision other)
{
        if(other.gameObject.CompareTag("Weapon") || other.gameObject.CompareTag("Bullet"))
        {
                dummyAnimator.SetTrigger("Death");
                GetComponent<BoxCollider>().enabled = false;
        }
}
```

The method `ActivateDummy()` raises the `Activate` trigger that starts the alert animation:

```
public void ActivateDummy()
{
        dummyAnimator.SetTrigger("Activate");
}
```

It is called from `EnemyCharacterTrigger`, which owns the activation zone. When the player enters that zone, the class iterates over its `targetEnemy` array and activates every agent it holds – again a one-to-many relation, one trigger to many agents:

```
private void OnTriggerEnter(Collider other)
{
        if(other.gameObject.CompareTag("Player"))
        {
                foreach(GameObject enemies in targetEnemy)
                {
                        enemies.GetComponent<TargetDummy>().ActivateDummy();
                }
        }
}
```

Because each class depends only on the interface it actually uses, agents and activation zones can be added to a scene by duplicating prefabs rather than by extending the code.

4. Discussion and conclusion

This study presents a working prototype of a VR shooter scenario that brings low-polygon modeling, skeletal animation, and interactive event handling into a single development pipeline. Blender-based character creation and Unity XR integration together produce an environment in which the user engages a dynamic agent through gesture-based input and spatial triggers, with animation responding to user-driven events in real time. The finite-state machine over baked clips gives deterministic transitions, and the UML model of the four runtime classes keeps the codebase small enough to extend without re-reading it in full.

The result meets the objective the work set out with: a reproducible and technically coherent route to interactive character content in VR, assembled entirely from open-source and freely available tools.

The pipeline is documented at the level of detail – modifier settings, rig generation, clip inventory, Animator triggers, class responsibilities – that a reader would need in order to rebuild it, which is what makes it usable as teaching material as much as a research substrate.

The scope of the prototype is nonetheless narrow, and several limitations should be stated plainly. No quantitative evaluation was carried out: the paper reports no frame-rate, latency, or tracking-accuracy measurements and no usability study, so statements about responsiveness rest on qualitative observation during development. The scenario comprises a single scene with one reactive agent, which exercises the architecture but does not stress it. Most interaction testing was performed with the XR Device Simulator on a desktop machine rather than on headset hardware, leaving headset-specific concerns – comfort, tracking loss, rendering cost – unexamined.

These limitations set the agenda for further work. Agent behavior could be enriched beyond the three-state machine used here, with behavior trees or hierarchical state machines supporting more varied non-player character responses [25]. Runtime IK solvers would let posture adapt to live player input rather than replaying baked clips, which is the natural next step given that IK is at present confined to authoring. The scene could be extended to several agents with coordinated, AI-driven behavior [26]. Finally, an empirical study is needed to quantify presence, motion precision, and interaction efficiency across different VR configurations, and to put numbers where this paper offers a demonstration.

Acknowledgments

This article is based on the thesis of Yuliia Sapsa, prepared under the academic supervision of Dr. Oleksiy Sinkevych and reviewed by Dr. Oleksandr Storozhuk. The work was carried out at the Department of Computer Science of the Ukrainian National Forestry University and completed remotely while Yuliia Sapsa was living and studying abroad in Spain. The authors are grateful to the open-source community behind Blender and Unity, whose tools made it possible to develop and test the virtual reality prototype described here.

Declaration on Generative AI

This publication contains sections of text edited using OpenAI's ChatGPT generative artificial intelligence tools for linguistic improvement. The underlying research, system architecture, code base, visual content, and experimental results were developed and refined entirely by the authors without the use of generative artificial intelligence for content generation or code synthesis.

References

[1] M. Slater, M. V. Sanchez-Vives, Enhancing Our Lives with Immersive Virtual Reality, Frontiers in Robotics and AI 3 (2016) 74. doi:10.3389/frobt.2016.00074.

[2] J. L. Rubio-Tamayo, M. Gertrudix Barrio, F. García García, Immersive Environments and Virtual Reality: Systematic Review and Advances in Communication, Interaction and Simulation, Multimodal Technologies and Interaction 1 (2017) 21. doi:10.3390/mti1040021.

[3] I. Kartiko, M. Kavakli, K. Cheng, Learning science in a virtual reality application: The impacts of animated-virtual actors' visual complexity, Computers & Education 55 (2010) 881–891. doi:10.1016/j.compedu.2010.03.019.

[4] A. Braun, R. Rizzo, XR Development with Unity: A beginner's guide to creating virtual, augmented, and mixed reality experiences using Unity, Packt Publishing Ltd, 2023.

[5] S. A. Christian, Enhancing Virtual Reality Experiences with Unity 2022: Use Unity's latest features to level up your skills for VR games, apps, and other projects, Packt Publishing Ltd, 2023.

[6] E. Sheshina, Designing and building a three-dimensional environment using Blender 3D and Unity game engine, Bachelor's thesis, Lapland University of Applied Sciences, Rovaniemi, Finland, 2016. URL: https://www.theseus.fi/bitstream/handle/10024/120138/Sheshina_Evgeniya.pdf.

[7] D. W. F. van Krevelen, R. Poelman, A Survey of Augmented Reality Technologies, Applications and Limitations, International Journal of Virtual Reality 9 (2010) 1–20. doi:10.20870/IJVR.2010.9.2.2767.

[8] R. S. Johansen, Automated Semi-Procedural Animation for Character Locomotion, Master's thesis, Aarhus Universitet, Institut for Informations Medievidenskab, Aarhus, Denmark, 2009. URL: https://runevision.com/thesis/rune_skovbo_johansen_thesis.pdf.

[9] I. D. Horswill, Lightweight Procedural Animation With Believable Physical Interactions, IEEE Transactions on Computational Intelligence and AI in Games 1 (2009) 39–49. doi:10.1109/TCIAIG.2009.2019631.

[10] W. R. Sherman, A. B. Craig, Understanding Virtual Reality: Interface, Application, and Design, The Morgan Kaufmann Series in Computer Graphics, 2 ed., Morgan Kaufmann, 2018. doi:10.1016/C2013-0-18583-2.

[11] R. Hess, The Essential Blender: Guide to 3D Creation with the Open Source Suite Blender, No Starch Press, 2007.

[12] T. Adams, M. Davies, FBX Games Development Tools, in: M. Ansari (Ed.), Game Development Tools, A K Peters/CRC Press, Boca Raton, FL, 2011.

[13] P. Fleck, D. Schmalstieg, C. Arth, Creating IoT-ready XR-WebApps with Unity3D, in: Proceedings of the 25th International Conference on 3D Web Technology, Web3D '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 1. doi:10.1145/3424616.3424691.

[14] L. Ecole, W.-T. Kim, J.-S. Yoon, Unity: A Powerful Tool for 3D Computer Animation Production, Journal of the Korea Computer Graphics Society 29 (2023) 45–57. doi:10.15701/kcgs.2023.29.3.45.

[15] D. Aversa, C. Dickinson, Unity Game Optimization: Enhance and extend the performance of all aspects of your Unity games, Packt Publishing Ltd, 2019.

[16] A. J. Moshayedi, A. Kolahdooz, L. Liao, Unity in Embedded System Design and Robotics: A Step-by-Step Guide, Chapman and Hall/CRC, 2022.

[17] S. Blackman, J. Wang, Unity for Absolute Beginners, Apress, 2014.

[18] L. Pirhonen, Creating an Optimized 3D-mesh for a Stylized Low-poly Video Game Character, Bachelor's thesis, Tampere University of Applied Sciences, Tampere, Finland, 2024. URL: https://www.theseus.fi/bitstream/handle/10024/873926/Pirhonen_Liisa.pdf.

[19] L. Cappannari, A. Vitillo, XR and Metaverse Software Platforms, in: M. Alcañiz, M. Sacco, J. G. Tromp (Eds.), Roadmapping Extended Reality: Fundamentals and Applications, John Wiley & Sons, Ltd, 2022, pp. 135–156. doi:10.1002/9781119865810.ch6.

[20] A. Belec, Blender 3D Incredible Models: A comprehensive guide to hard-surface modeling, procedural texturing, and rendering, Packt Publishing Ltd, 2022.

[21] R. Parent, Computer Animation: Algorithms and Techniques, Newnes, 2012.

[22] C. Welman, Inverse kinematics and geometric constraints for articulated figure manipulation, Master's thesis, Simon Fraser University, Burnaby, BC, Canada, 1993. URL: https://viterbi-web.usc.edu/~jbarbic/cs520-s25/ik/welman.pdf.

[23] T. Norton, Learning C# by Developing Games with Unity 3D, Packt Publishing Ltd, 2013.

[24] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, The Addison-Wesley Object Technology Series, Addison-Wesley, 1999.

[25] Y. A. Sekhavat, Behavior Trees for Computer Games, International Journal on Artificial Intelligence Tools 26 (2017) 1730001. doi:10.1142/S0218213017300010.

[26] Y. Miyake, Character Artificial Intelligence, in: N. Lee (Ed.), Encyclopedia of Computer Graphics and Games, Springer International Publishing, Cham, 2024, pp. 285–293. doi:10.1007/978-3-031-23161-2_304.