Archery

Final Project for the course of Interactive Graphics, 2022

Simone Gennenzi - gennenzi.1848670@studenti.uniroma1.it

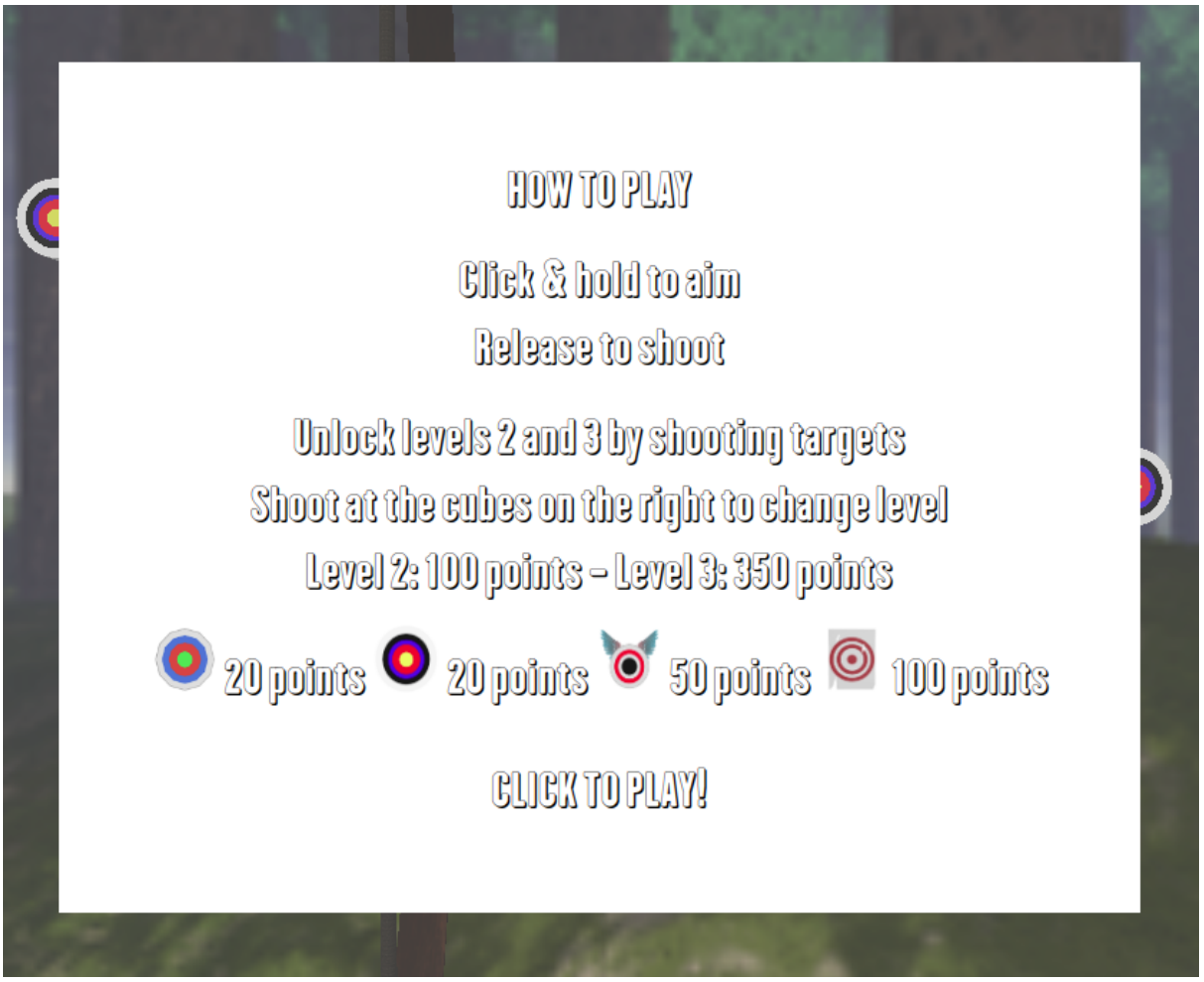
Giorgia Ristich - ristich.1839919@studenti.uniroma1.it

Dario Petrillo - petrillo.1839609@studenti.uniroma1.it

SapienzaInteractiveGraphicsCourse/final-project-archery

The game 3
Levels 4
User interaction 6
Game objects 7
Bow 7
Arrow 7
Menu cube 8
Simple target (2 types) 9
Winged target 9
Poster target 10
Crosshair 10
Scenes 10
Overlays 12
Libraries 13
Game architecture 14
Animations 15
Score 15
Simple Targets 16
Poster Target 16
Winged Target 16
Bow and arrow 17
Collision detection 18
Graphics post processing 19

The game



Archery is a first person arrow shooting game with different targets and three different levels. The player can press and hold to aim at the targets and release the mouse to shoot the arrow. There are four targets in the game with different point values:


20 points


20 points


50 points


100 points

To unlock the different levels, the player has to reach specific thresholds:

Level 2 → 100 points
Levels 3 → 350 points

Once a level is unlocked, a corresponding cube will appear on the right. These cubes are valid targets that, when hit by the arrow, change the level.

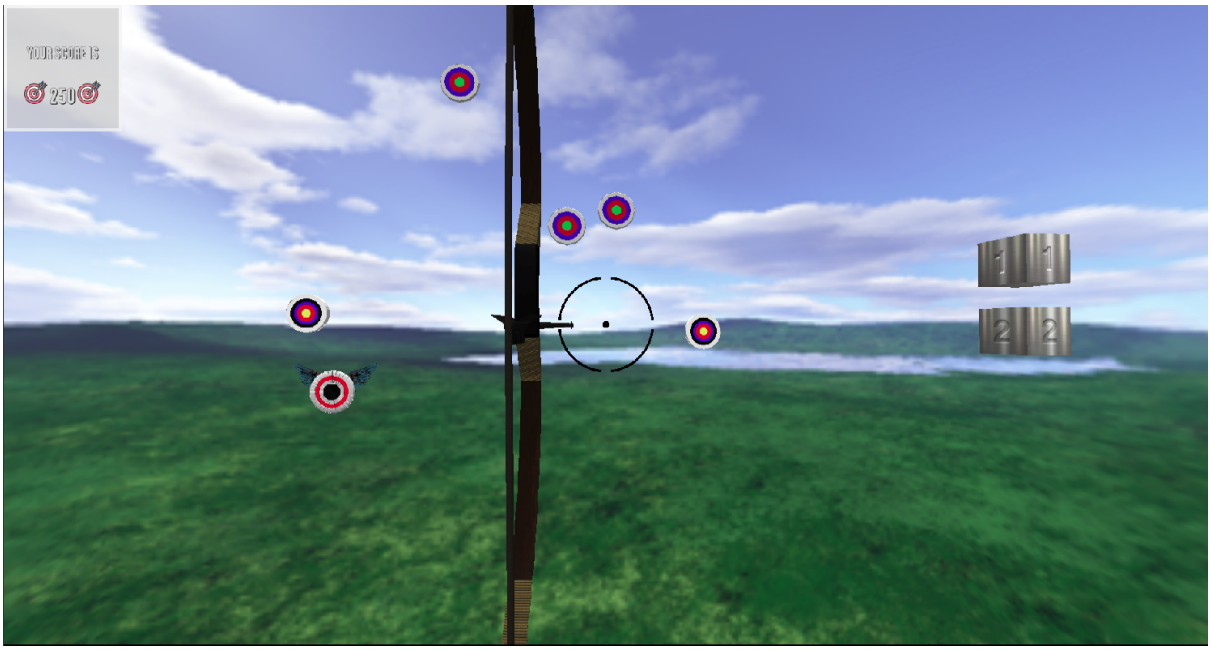
Levels

Level 1

This level consists of four simple targets which move horizontally and vertically with a low speed.

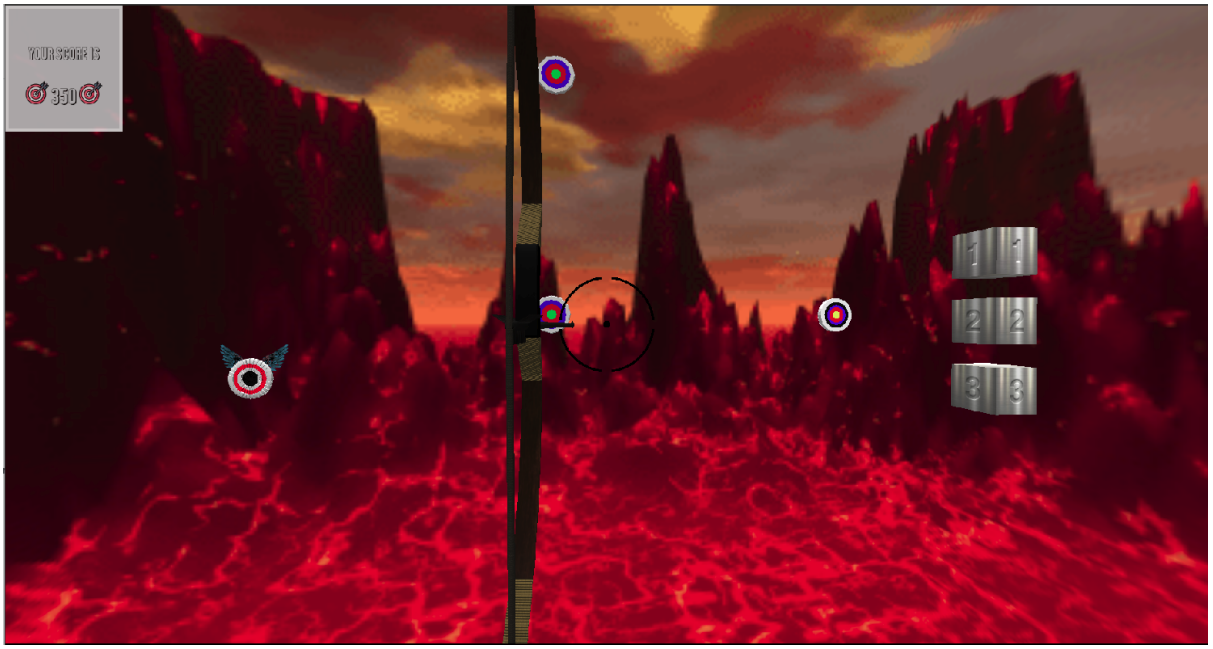
Level 2

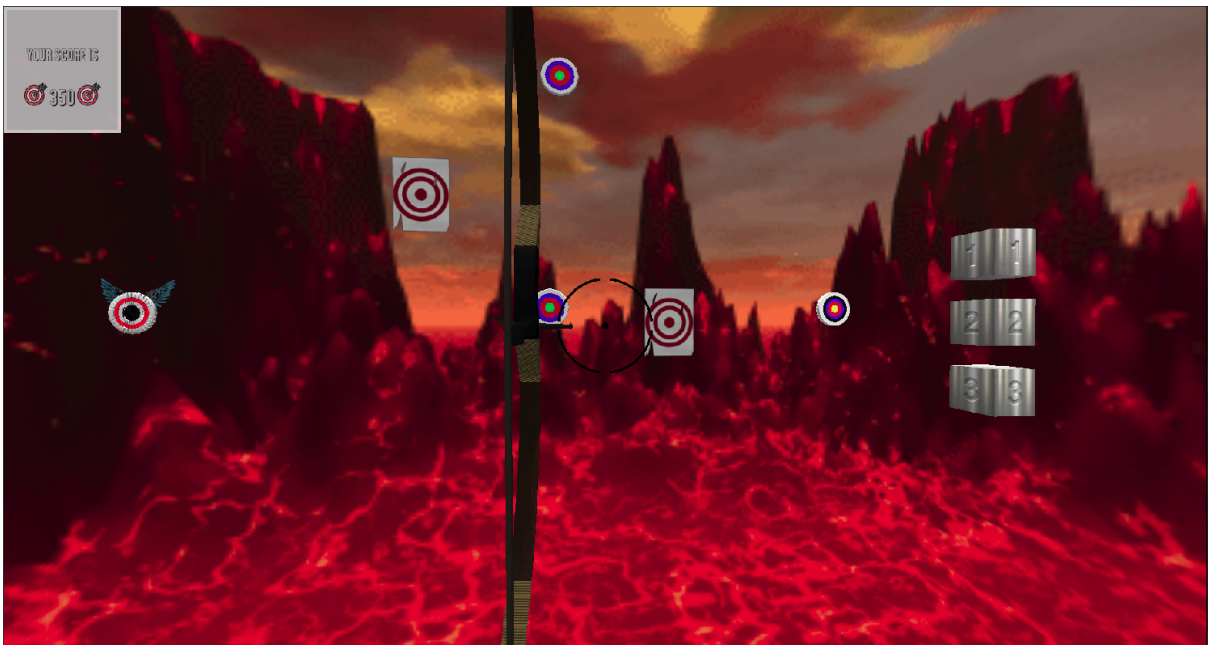
In this level the speed is increased (and so the difficulty) and a winged target which moves in a circle has been added.



Level 3

In the last level, the targets move faster and two poster targets appear after a timeout.



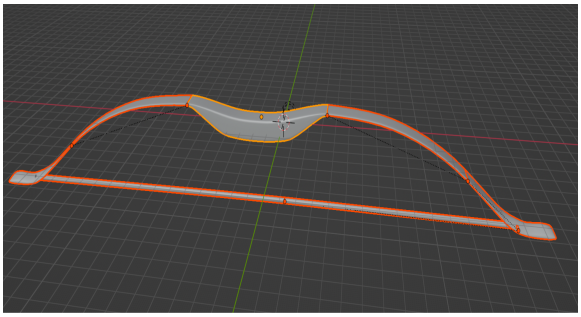


User interaction

The user interaction implemented in our project is based on the pointer lock control APIs, with Three.js' `PointerLockControls` abstraction. Through these controls, the user can look around, aim at targets and shoot, or he can shoot directly without aiming first. In addition to the four target types, the user can also point and shoot at the menu cubes on the right, which, when hit, change the active level. A welcome side effect of the pointer lock controls is that the mouse is hidden from the screen, giving a more immersive experience to the user. We replace the mouse with a crosshair model as a targeting aim.

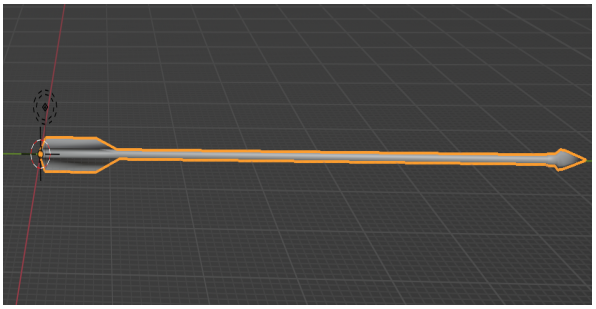
Game objects

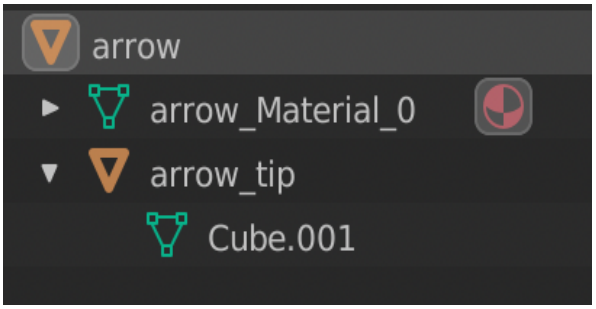
Bow



The bow was divided in seven parts with Blender, creating the hierarchical structure shown on the right. The root is the center of the wooden parts, and each of the sides is composed of two more pieces to show the wood flexing. The rope is split at its center, and is attached to the bottom side of the bow in the hierarchy. When animating, the center point of the rope is also used as the origin of the arrow.

Arrow

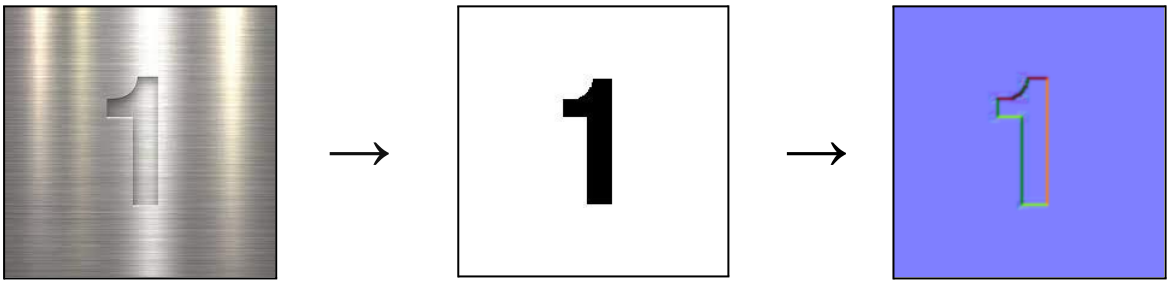




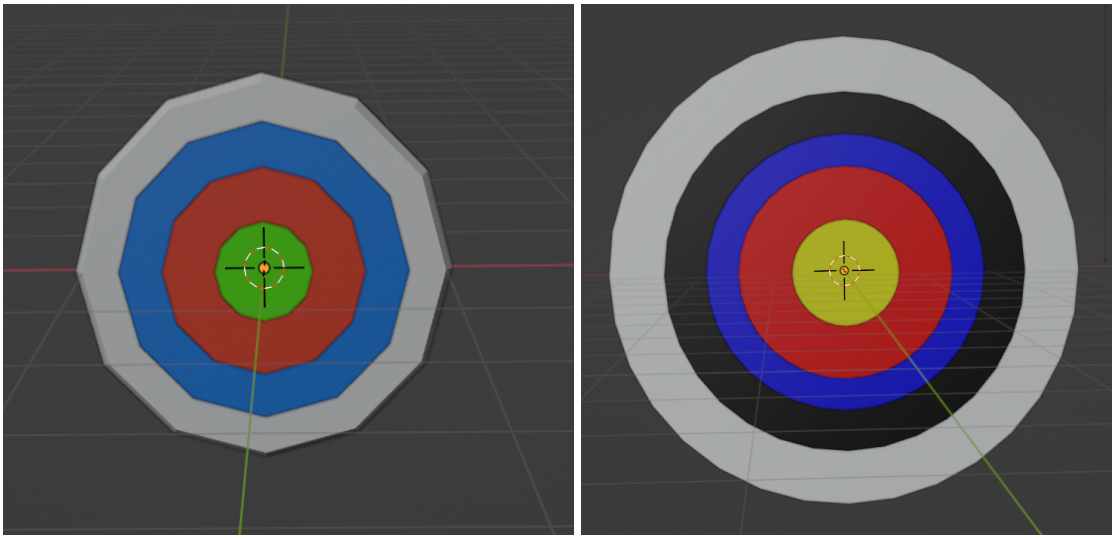
Like the bow, the arrow was imported in Blender. However, in this case the arrow is composed of two parts: the arrow and its tip (see image on the right). The tip is a small cube hidden inside of the arrow's tip, and its position, automatically updated when the model moves, is used to detect collisions.

Menu cube

The cubes have both a color texture and normal mapping applied, to make the numbers as visible as possible to the player.
The color textures were first converted to depth mappings semi-manually (using `utils/height_map.py`), and the resulting images were used to generate normal maps using an online tool (https://cpetry.github.io/NormalMap-Online/). The steps for the “1” texture can be seen below, and “2” and “3” were generated in an analogous way.

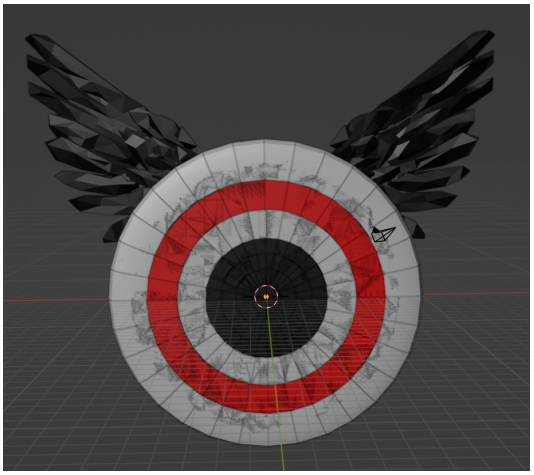


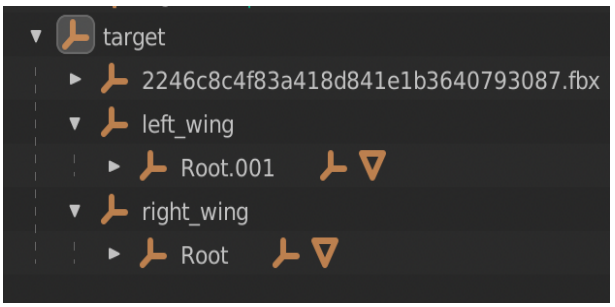
Simple target (2 types)



The two models of simple targets can be found in all three levels.

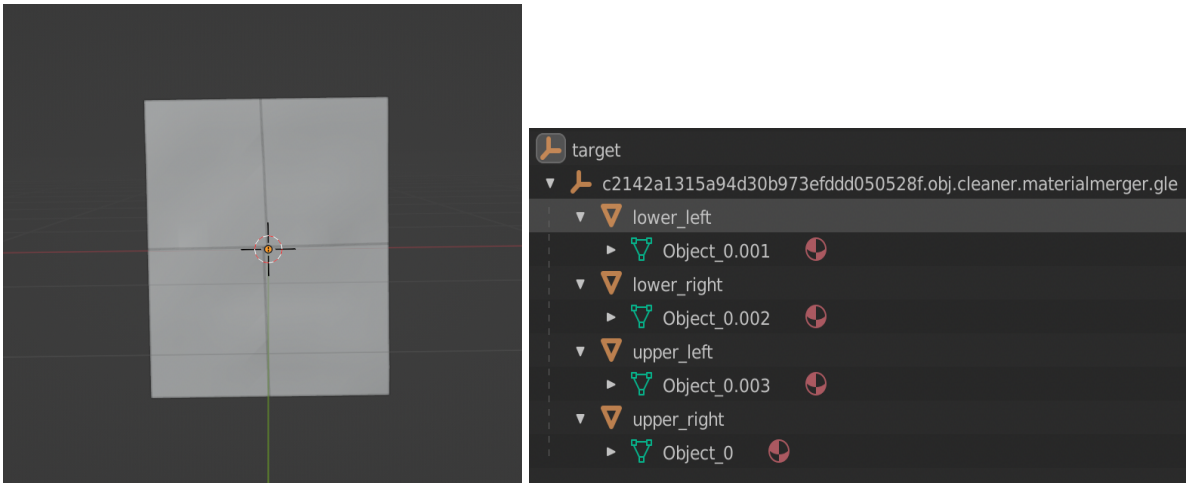
Winged target





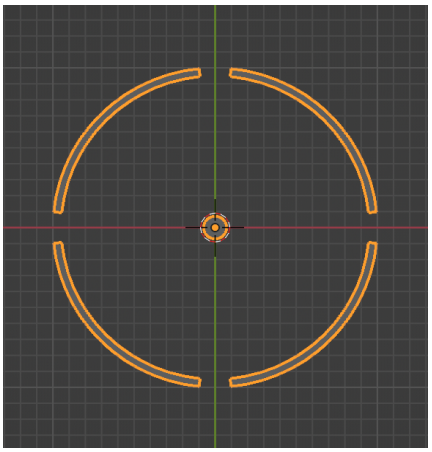
The winged target is a hierarchical model which we obtained by adding the two wings to the target 3d model in Blender. The root of the hierarchy is the target itself and its children are the two wings: left_wing and right_wing (see image on the right). The wings are animated and together with the target's circular movement they simulate a flying object (see `Animations` section).

Poster target



In this case, we divided the target in four parts composing a hierarchical model, which are then animated when the target is hit.

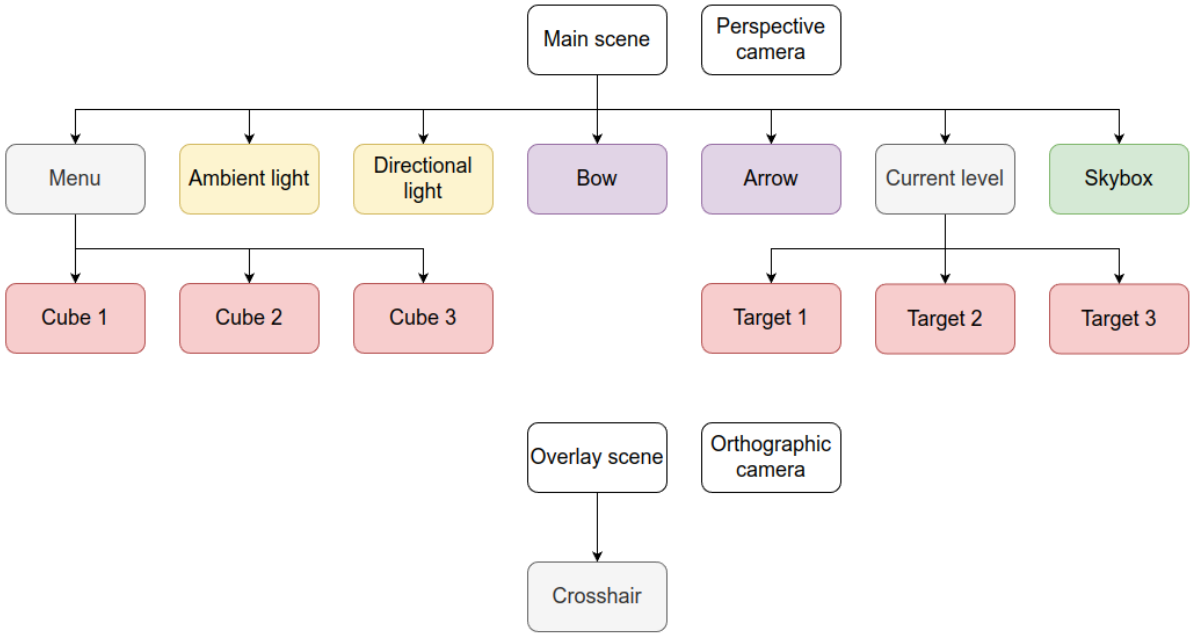
Crosshair



The crosshair is displayed in an overlay to help in aiming at the targets.

Scenes

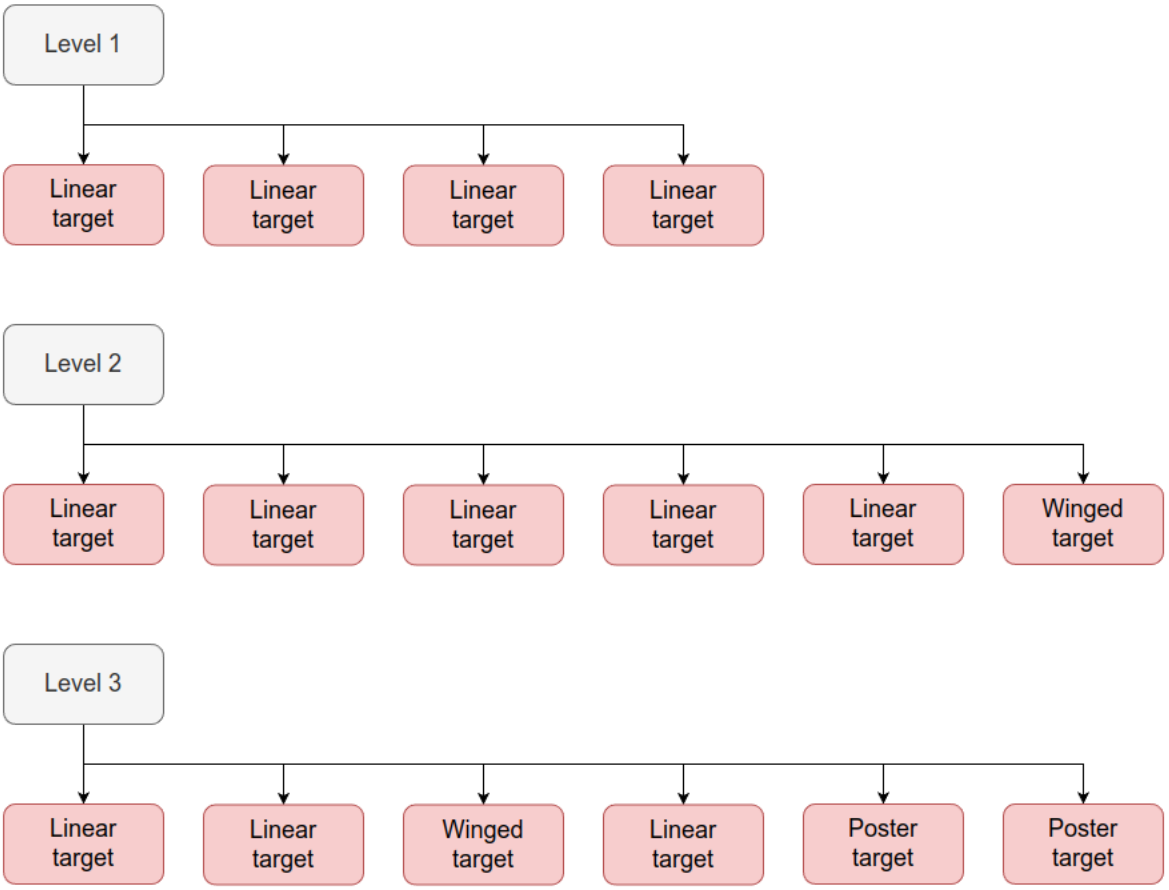
The main scene contains most of the 3d objects, the lights and the skybox texture. A separate overlay scene is used with an orthographic camera to render the crosshair on top of the game, keeping it in the center at all times.



Objects are color coded:

- Gray: Object3D
- Yellow: Light
- Green: Texture
- Purple: GameObject (see Game Architecture section)
- Red: CollidableObject (see Game Architecture section)

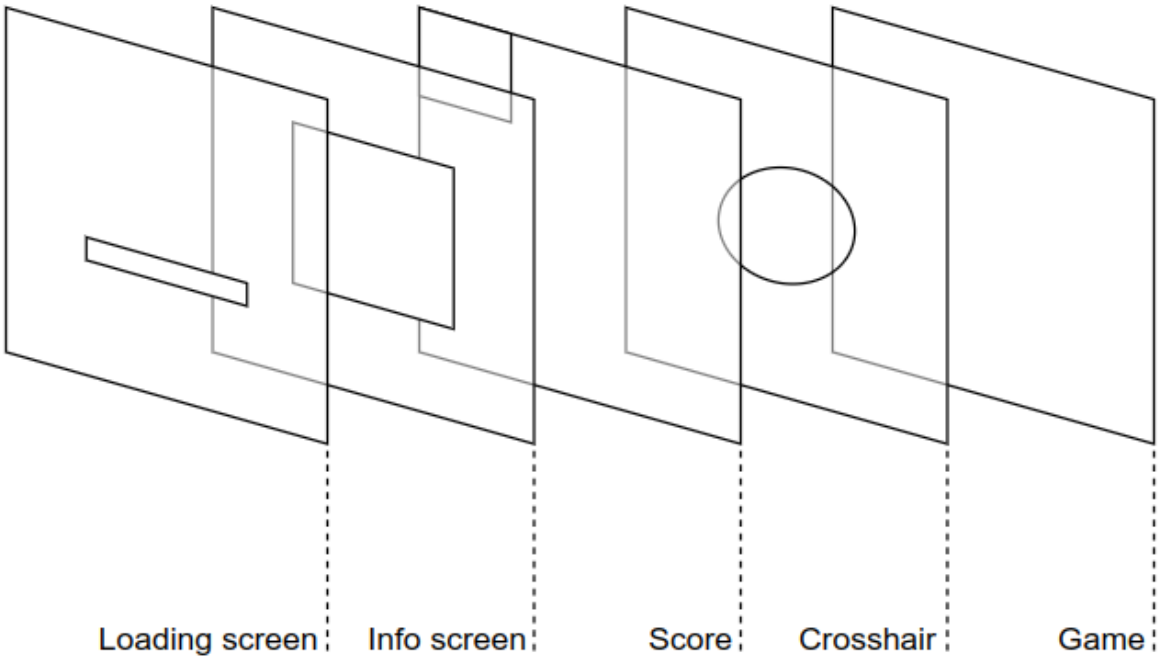
Since the differences between levels are limited to what targets are present and their animations, we keep the targets of each level in an Object3D which is inserted in the main scene (marked “Current level” in the scene diagram above). When the user changes level by hitting the corresponding menu cube, we change the skybox and the Object3D containing the targets with the new level’s ones.



See the Game architecture section for a description of the different target types.

Overlays

The game contains multiple layers, organized as follows



The top three layers are defined in HTML and contain the UI of the game:

- `Loading screen`: shown at the start, while the assets are being downloaded.
- `Info screen`: covers the whole game when pointer lock is not acquired, and displays instructions on how to play.
- `Score display`: shows the current score. This layer is also used to display the "Level X unlocked" notification when a new level is unlocked.

The final two layers are rendered inside the WebGL canvas:

- `Crosshair`: This scene is rendered with an orthographic camera, and displays a crosshair symbol to help the player with aiming. It is kept in the middle of the screen at all times.
- `Game`: This final layer contains all the 3D entities of the game.

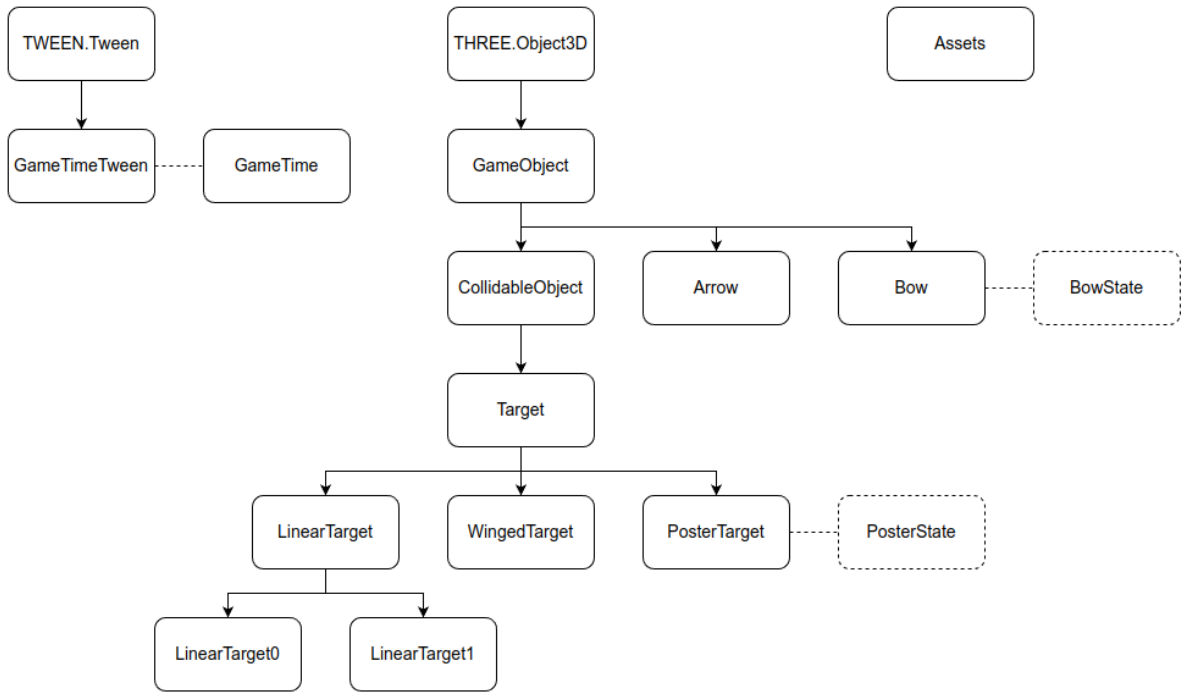
Libraries

This project uses the following libraries

Library	Description
Three.js	WebGL 3D graphics
Tween.js	Animations
es-module-shims	Importmap compatibility layer for older browsers; has no influence on the game graphics but allows for it to run on more browser versions

Game architecture

We implemented the following structure for the game, which can roughly be subdivided in three parts: animations, game objects and asset management.



Animations are implemented via `GameTimeTween` (*js/GameTimeTween.js*), a light wrapper around `TWEEN.Tween` that keeps track of game time instead of real time. This is done in order to allow for pausing the game without the animations suddenly

jumping when the game is resumed, and additionally makes the animations smoother since they all rely on the same timestamps at every frame. If we were to use Tween directly, each animation would have a slightly different timestamp depending on the exact time at which it was started.

The main class hierarchy is the one used for game objects. The basic class, `GameObject` (*js/GameObject.js*), implements lookup functionality for children of hierarchical objects, so that when animating we can lookup individual components by name in the `GameObject.parts` dictionary. Since the arrow and bow are never targets of a collision, they inherit directly from `GameObject`.

Support for collisions is implemented in `CollidableObject` (*js/GameObject.js*), which has the responsibility of calculating the convex hull of objects on their creation. It also keeps track of their initial position to compensate for animations when doing collision detection - see the Collision Detection section for more details.

Deriving directly from `CollidableObject`, the `Target` class keeps track of the active animations of a target, and each different subclass implements the specific functionality of a type of target. In the case of `LinearTarget`, the two subclasses use different 3D models, and for `PosterTarget` we keep track of its current state via the `PosterState` enum.

The final component is the `Asset` class, which together with the loading code in main.js loads all the assets required for the game while showing a loading screen. The rest of the code can then access the assets directly, with the assumption that they have all been loaded at game startup.

Animations

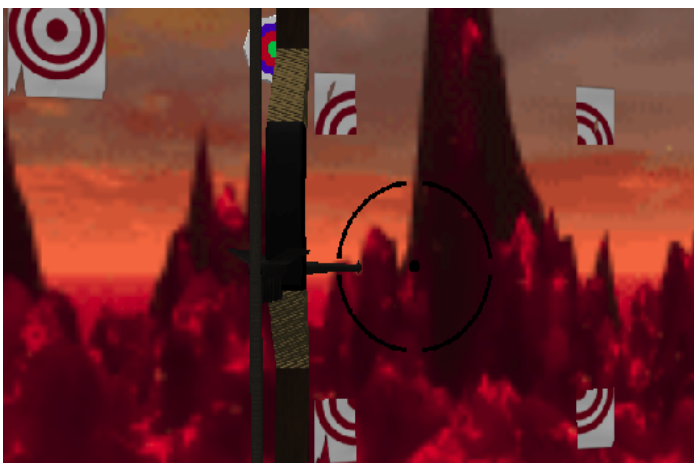
Score

In order to increase the visibility of a collision, the score is animated, so it's updated every time an object is hit. This gives visual feedback to the user whenever a collision occurs.

Simple Targets

The animation of these targets consists in an horizontal or vertical movement along the related axis from their initial position to a specified point.

Poster Target

The animation of this target is governed by a state machine as follows:

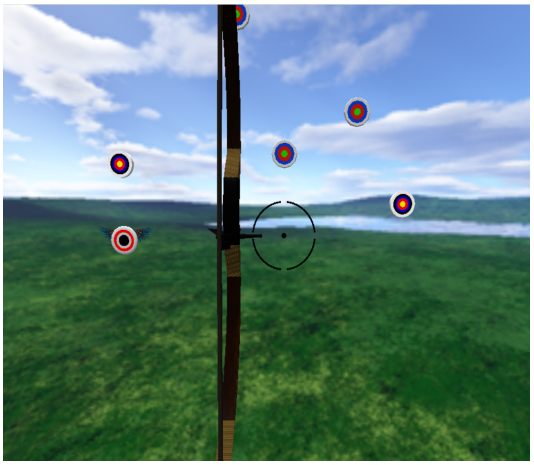
- At the beginning of the level, the poster is hidden and in its default state `Waiting`
- After a 5 second timeout, it changes to `Active` and is shown in the scene
- Then, when the poster is hit, its state becomes `Hit` and each of the four parts of its hierarchical model start moving diagonally outwards.

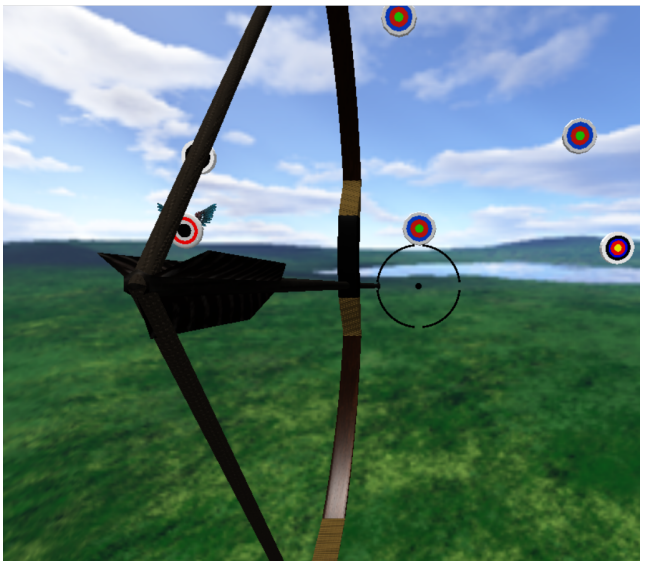
Winged Target

For this target, we can notice two animations: the `first` related to the rotation of the wings and the `second` related the circular movement of the entire hierarchical model of the target.
The two wings are rotated back and forth (through the yoyo functionality) by an angle of thirty degrees in order to simulate the flight.
Instead for the movement of the entire target:

- On the x-axis, the target is moved back and forth on the diameter of the circle.
- On the y-axis, the movement is divided in three sub-animations chained, see more details on `Targets.WingedTarget` in `Js/Target.js`

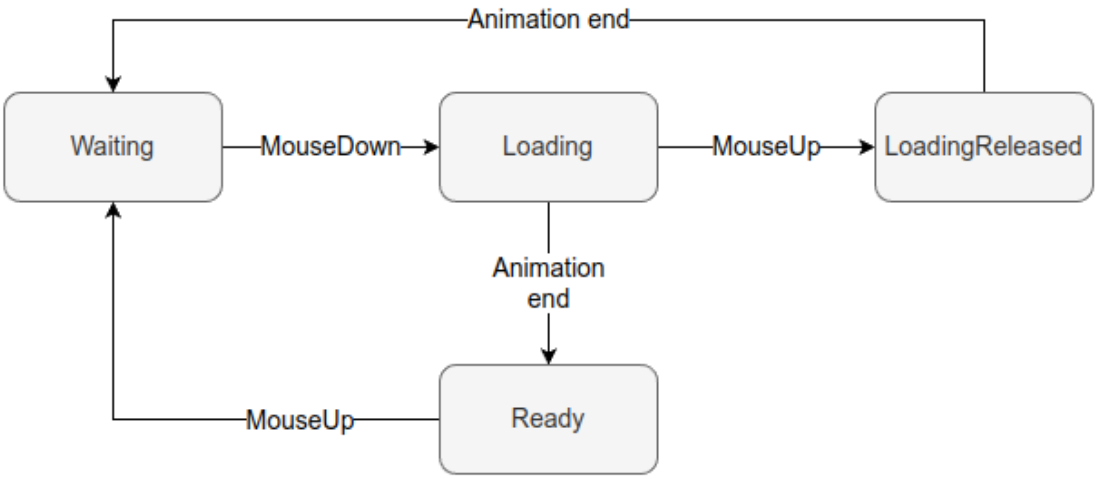
Bow and arrow





The animation of this target is governed by a state machine as follows:

- The bow starts in state `Waiting`
- On a `mousedown` event, the animation to draw to bow starts, and the state changes to `Loading`. The animation exploits the hierarchical structure of the bow when deforming the wooden section, and has an additional calculation to keep the rope aligned to the rest of the model. This is necessary to keep the rope attached to the bow at both its ends, and its implementation can be found as `Bow._updateRope` in `js/Bow.js`. Additionally, the arrow is moved together with the bow as if it were part of the same hierarchical model.
- When the animation ends, the bow moves to `Ready` and waits for the player to release the mouse.
- From `Ready`, when a `mouseup` event is fired the animation to release the bow is started, and the arrow is launched. The state changes back to `Waiting`.
- If instead the user releases the mouse while the animation is still running, the state changes to `LoadingReleased`, and the animation continues to run.
- From `LoadingReleased`, when the animation ends we fire the arrow, start the bow closing animation and move back to `Waiting`.



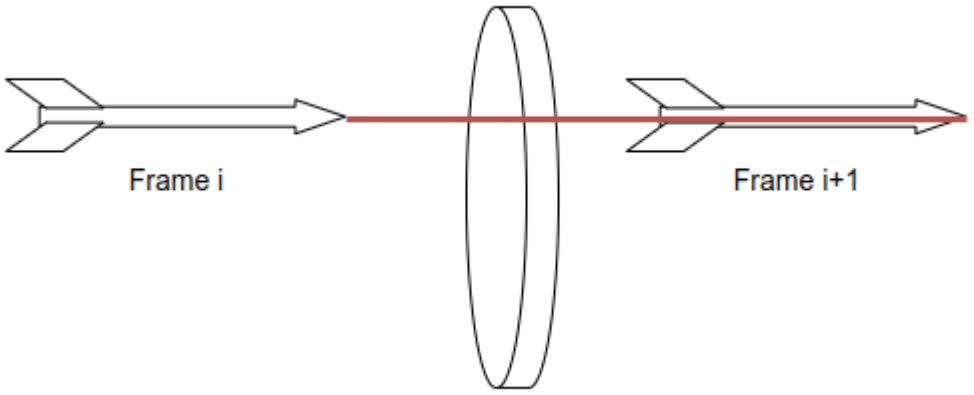
The goal of this state machine is to make the whole animation sequence more realistic, by forcing the bow to deform fully before launching the arrow even if the user releases the mouse early.

Collision detection

The targets pose two distinct challenges for collision detection:

1. They are mostly round objects. A simple bounding box model would detect a collision in the corners even if there is no part of the model there.
2. They are relatively thin. The arrow could move from one side of an object to the other in a single frame, avoiding detection if object-to-object intersection is implemented.

To address these issues, we use the target's convex hull as an intersection target, and we consider the intersection between the target and the line segment traced by the arrow between the last and current frame to detect a collision.
In order to avoid calculating a general polyhedron-polyhedron distance at every frame, we only consider the tip of the arrow and whether it went through a given object in the current frame. This is realistic enough for the game, as cases where the body or back of the arrow hits a target are very rare and would only happen at the very edges of a target.
Additionally, calculating a 3D convex hull is a slow operation, so we only do it once when the models are loaded. Since the target animations are mostly translations, when we look for an intersection with the arrow segment we can translate the segment in the opposite direction to get the correct answer.



For more details, the collision detection implementation can be found as `Arrow.checkCollision` (*js/Arrow.js*) and `CollidableObject` (*js/GameObject.js*)

Graphics post processing

We include some post processing for edge highlighting of the menu cubes. This makes it clear to the player that the objects are to be interacted with, and is implemented with an `EffectComposer` chain in Three.js. The chain includes a `RenderPass` to render the main scene, followed by an `OutlinePass` which highlights a set of objects:

```
    const composer = new EffectComposer(renderer);
    const renderPass = new RenderPass(scene, camera);
    const outlinePass = new OutlinePass(new THREE.Vector2(canvas.width,
canvas.height), scene, camera);
    composer.addPass(renderPass);
    composer.addPass(outlinePass);
```

The highlighted cube is selected via ray casting from the center of screen position, and given to the outline pass:

```
   const raycaster = new THREE.Raycaster();
   function render(time) {
   [...]
     raycaster.setFromCamera({x: 0, y: 0}, camera);
     const intersects =
raycaster.intersectObjects(levelSelector.menu_cubes);
     if(intersects.length > 0) {
       const new_cube = intersects[0].object;
       outlinePass.selectedObjects = [new_cube];
     }
     else {
       outlinePass.selectedObjects = [];
     }
   [...]
   }
```