Guideline for the GameJam 2025-01 of Fachschaft Mathematik/Physik/Informatik

2025-01-20

Contents

1 Organisation

- The GameJam starts on 10.01. at 16:30. The final version has to be submitted until 12.01. 18:30.
- The games are rated by you until 15.01. 14:00. Based on the participants' preselection, you will present the games to the jury on 15.01. from 15:00 to 17:00. The jury will then choose the winner.
- During the presentation, you will show the games on your own devices. You shall further upload a compiled version, that is distributed to all participants.

2 Technical Requirements

- The game has to support either Windows (10+) or Ubuntu (24.04+). If possible (your engine supports this, and you used no OS-specific APIs), you shall submit versions for Windows, macOS, and Linux.
- If your game is intended for gamepads, it must also be possible to play one player with keyboard and mouse.
- There will be repositories in the RWTH Gitlab. There you submit your final version.
- We do not mandate a specific engine, though we do recommend to use Godot or Unity. We also recommend that you get familiar with the engine, you would like to use, before the jam starts.
- You may only use assets and libraries that are available free of charge, and only if the license permits you and us to distribute your unmodified binaries to the other participants. To keep track of the

licensing, we recommend creating a list of all used assets and their source, including assets that don't require attribution.

3 Introduction: How to develop games?

3.1 What do I need for a game?

To create a game several components are usually needed:

2D Graphics To show something in the game, you will need graphical assets. In 2D games, the graphics consist of 2D UI Overlay elements like the menu, buttons, or the score. The rest of the game graphics are also 2D images and are usually called *sprites*. Often, many sprites are stored together in the same file called spritesheet or texture atlas. Animations are done by having different variants of an object and cycling through them.

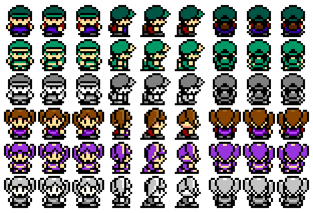


Figure 1: Example character sprites with walk animations from Corey Archer

3D Graphics In 3D games, the UI overlay is done the same way as in 2D games, but the 3D world is more complicated.

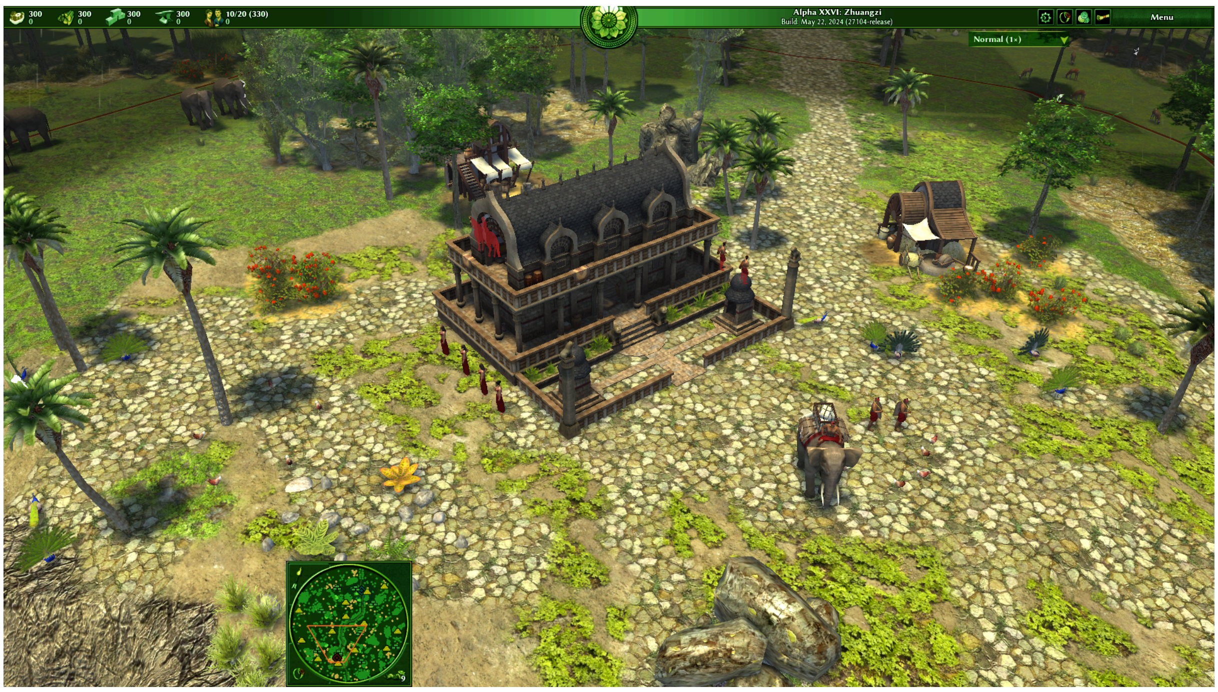


Figure 2: A screenshot from 0.A.D.. The 2D UI Overlay is marked green. Buildings, People, Animals, Ground, Trees, and everything else are 3D models.

3D models are usually created in dedicated 3D modelling software like Blender. The components are shown in Figure 3. The basis is a mesh (usually consisting only of triangles). Next we need textures: Each vertex (point) in the mesh has a UV-map attribute, that tells to which point in the 2D texture it corresponds. There can be several textures. The most common textures are *diffuse* texture (what color does the point have), *normal* map (in which direction is light emitted), *specular* map (how much does the reflection use the light color instead of the diffuse color), but there are several other types. Models also have a material, with many more properties to change its appearance. You can see the influence of many common properties in these examples for common properties. Additional documentation including rarer properties are in the blender documentation for physically based rendering (PBR). Note, that advanced shadered materials from 3D software are usually not exportable to game engines. This is in part because of lack of standardized formats, but also because games require real-time rendering, which heavily restricts how complex graphics can be.

For inanimate objects, this is sufficient. For animated objects we need some more components. First, we add bones to the model (this process is called *rigging*). Then we say for each combination of bone and vertex, how much a change in the bone position should influence the position of that vertex (this process is called *skinning*). Then for each point and each keyframe, we say what position the bone is in that frame. This way of animation is called *skeletal animation*. If the relative position between bones is fixed, the connection is called a *joint*.

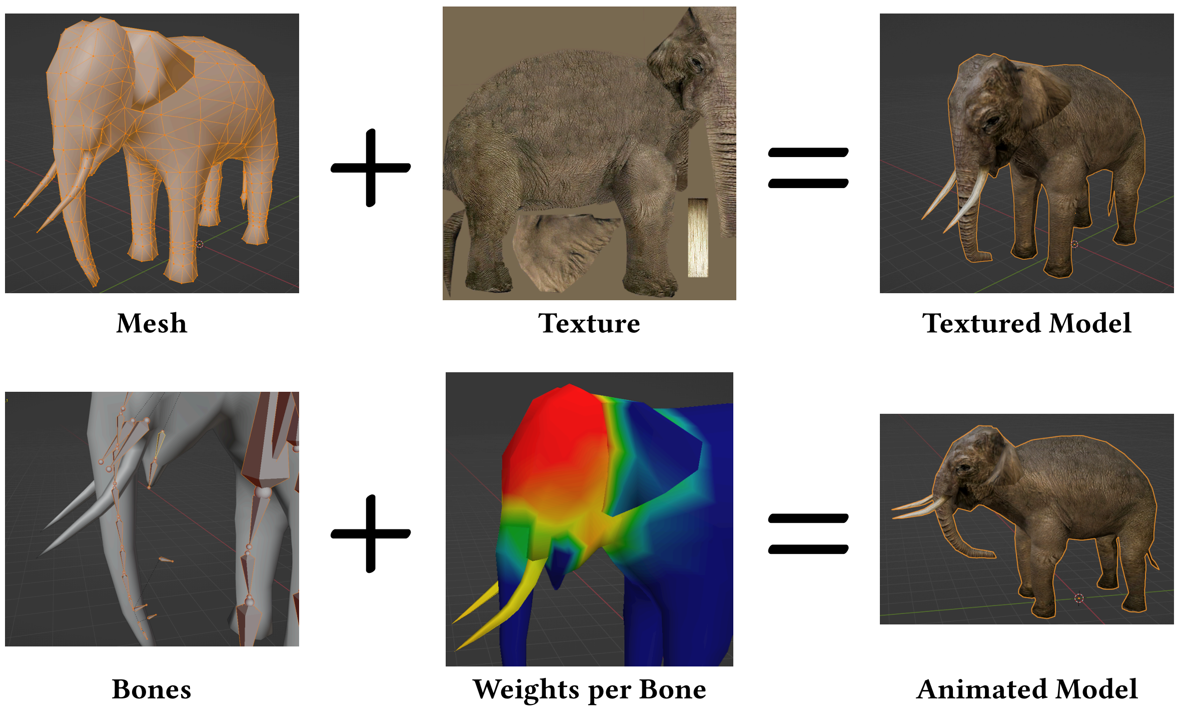


Figure 3: Typical components of a 3D model. Example assets from 0.A.D.

Sound Other assets needed are sounds. These are divided into music (in minecraft that would be the background music or the menu music) and sound effects (short: *SFX*) (f.e. the placing of a block, the opening of a door, the click on a menu button). There is usually one music played concurrently, while sound effects are triggered by events and may overlap without being faded into each other.

Physics Most games need some form of physic simulation. For this, each entity that should be influenced by physics, gets a component (most commonly a rigid body component). The components tell the weight of the object, whether its position or rotation is fixed, how much it bounces (*bounciness*) or is affected by friction (*roughness*).

A physics engine simulates in every frame the physical changes to all entities with physical components. It applies forces, moves objects, determines collisions, resolves collisions by calculating impulses, and sends collision events to the object's scripts. Physics other than rigid body physics (f.e. soft body physics or fluid simulation) are more difficult to use in games. In both Godot and Unity, colliders can be generated from arbitrary meshes. However, it is much more efficient to use one or multiple simple shapes (spheres, cubes, capsules or other built-in shapes) as colliders for your entities.

Even games with no physics can need colliders, f.e. to determine which object is selected on a mouse click. The task of determining what colliders a ray (a line going from a point in one direction) will intersect is called *Ray Casting*. When trying this for a collider, it is called *Shape Casting*.

Game engine The game engine loads required assets on startup and then enters the main loop. In every frame, the loop will go through several stages (usually this includes `PreUpdate`, `Update`, and `PostUpdate`), giving scripts the opportunity to act in these stages. There is also a special stage called `FixedUpdate`, that is called a fixed number of times per second, independently of the game's frame rate. After each iteration of the main loop, the engine will render the world to the screen. During rendering, the engine might also perform post-processing steps like antialiasing[1].

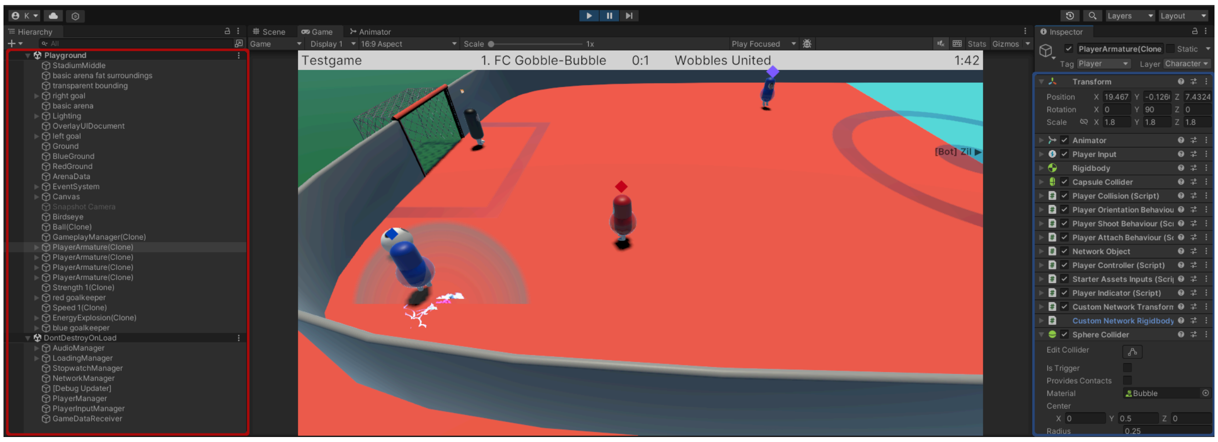
Networking For a 2-day game jam you should probably stick to the default networking method of your game engine. However, if you want to scale and have to consider latencies in a fast-paced game, you would have to put a lot of thought and work into networking. Some considerations are collected here. If you want to do your own networking, I also recommend the networking section of the GDC presentation from the RocketLeague developers on how to design physic-based games.

3.2 Entity-Component-System (ECS)

This section does not apply to Godot, which uses an object-oriented class hierarchy and not ECS.

Game engines sometimes use a design pattern named entity-component-system (ECS)[2]. An entity is like an object. It could be a physical object in the game, f.e. the player character, the camera, a light source, a spawn point, or a projectile. But it could also be something without relevant 3D Location, f.e. an event receiver or a global script. Entities have components, that define the entities' functionality.

For example, an arrow projectile could have the following components:
- a transform component, that tells where the entity is located in the world and how it is rotated
- a mesh component with the mesh that should be rendered at the entities position
- a collider, that is used by the physics engine to determine whether the arrow hit some other entity
- a rigid body component, that tells the physics engine to move the entity according to rigid body physics
- a script component, to process events (f.e. the event that the arrow hit another entity)



PlayerArmature(Clone) Static
Tag Player Layer Character

Transform
Position X 19.467 Y -0.126 Z 7.4324
Rotation X 0 Y 90 Z 0
Scale X 1.8 Y 1.8 Z 1.8

Animator
Player Input
Rigidbody
Capsule Collider
Player Collision (Script)
Player Orientation Behaviou
Player Shoot Behaviour (Sc
Player Attach Behaviour (Sc
Network Object
Player Controller (Script)
Starter Assets Inputs (Scri
Player Indicator (Script)
Custom Network Transforn
Custom Network Rigidbody
Sphere Collider
Edit Collider
Is Trigger
Provides Contacts
Material Bubble
Center X 0 Y 0.5 Z 0
Radius 0.25

Figure 4: Screenshot from Unity with the hierarchy of entities to the left (the red box) and the components of a PlayerArmature to the right (the blue box).

[1]See https://docs.godotengine.org/en/stable/tutorials/3d/3d_antialiasing.html for Godot's options

[2]Strictly speaking Unity does not use ECS either, but only the EC-pattern. The systems part is not used in Unity.

3.3 Modularization

Like in almost every programming language, it is helpful to group functionality into modules. For entities, you might have a player entity with several colliders as children, an item in the hand, a nameplate on top and components for animation, physics, remaining powerup durations, or input processing. To avoid recreating this for every player from scratch, you should instead make a scene for the player figure, and instantiate it for every player. In Godot you just create a regular scene and instantiate it in another scene. In Unity this not possible. Instead, there are special scenes called *Prefabs* that can be instantiated in regular scenes.

3.4 Where can I obtain free assets?

3.4.1 Blendswap

Blendswap is a public exchange platform for blender files, containing over 25, 000 blender files under (mostly) free licenses. It is mainly useful to obtain meshes. Sometimes the materials and textures are useful too, but not every blend has textures and not all materials are usable in game engines.

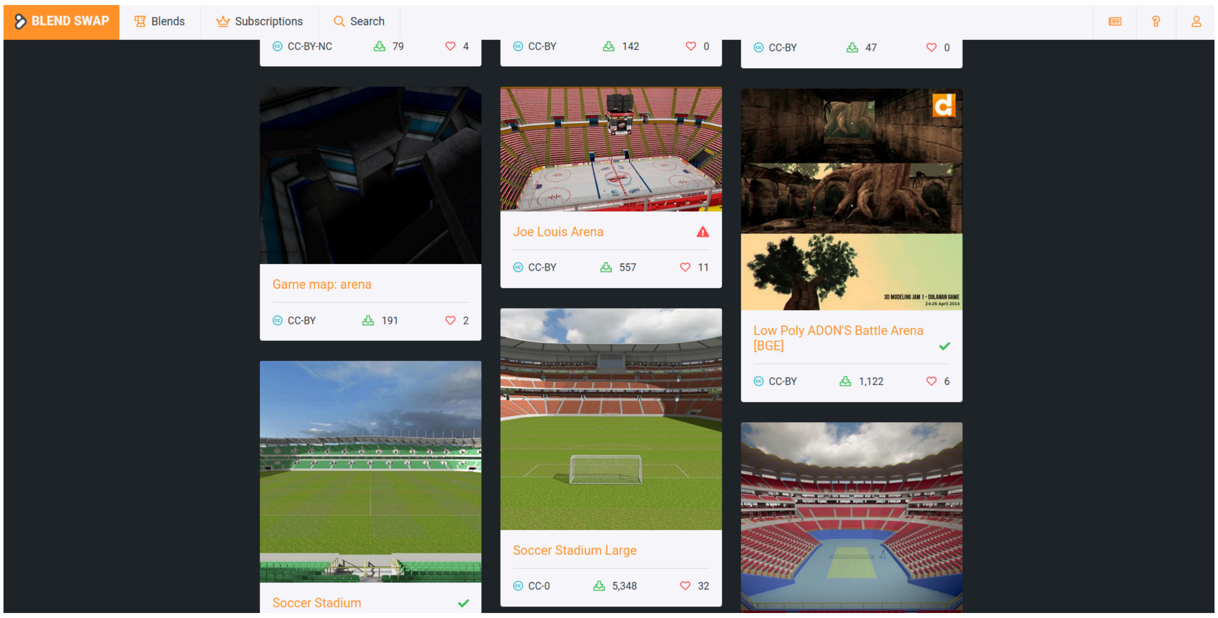


Figure 5: Blendswap search results when searching the term “arena”

3.4.2 Kenney

Kenney creates low-poly 3D assets, simple 2D assets and UI elements, and a few sound effects, and gives them to the public domain.

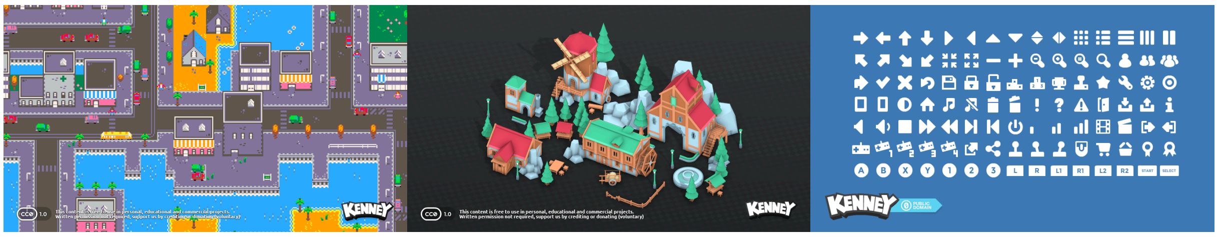


Figure 6: Examples for scenes created with Kenney’s graphics

3.4.3 OpenGameArt

OpenGameArt is a public exchange platform for game assets with free licenses. It contains 2D and 3D assets, particle effects, music, and sound effects. A difficulty when using OpenGameArt, are the different artistic styles used between the many graphics.

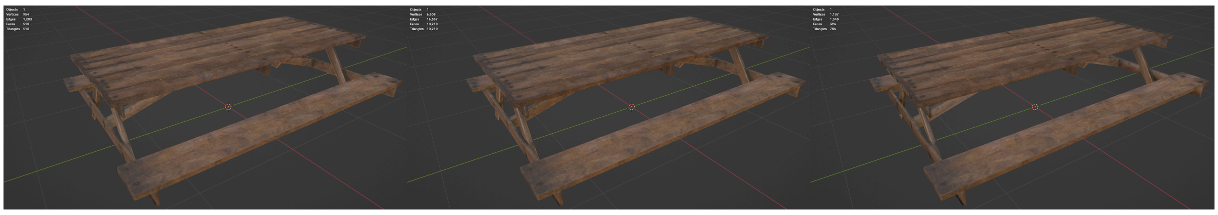
Figure 7: Examples for scenes created with OpenGameArt assets

3.4.4 Polyhaven

Polyhaven is a collection of high-resolution textures and high-poly 3D models.

Figure 8: Models from polyhaven

The assets from polyhaven usually have many polygons. To lower hardware requirements and computation effort, you might want to apply the decimate modifier in blender to reduce polygon count with minimal distortion.



Collapse (ratio = 0.05)	**Original**	**Planar (angle limit = 80°)**
510 faces	**10210 faces**	**394 faces**

Figure 9: Picnic Table from polyhaven with different decimate settings

3.4.5 freesound.org

freesound.org is a platform, containing over 600, 000 sound effects and ambient sounds under (mostly) free licenses.

3.4.6 0.A.D.

There are many 3D models of ancient buildings, animals with animations, and plants, developed for the open-source game 0.A.D., available under CC-BY-SA 3.0, and downloadable from https://github.com/0ad/0ad[3].

However, while high-quality and made for games, the animations can be difficult to integrate technically.

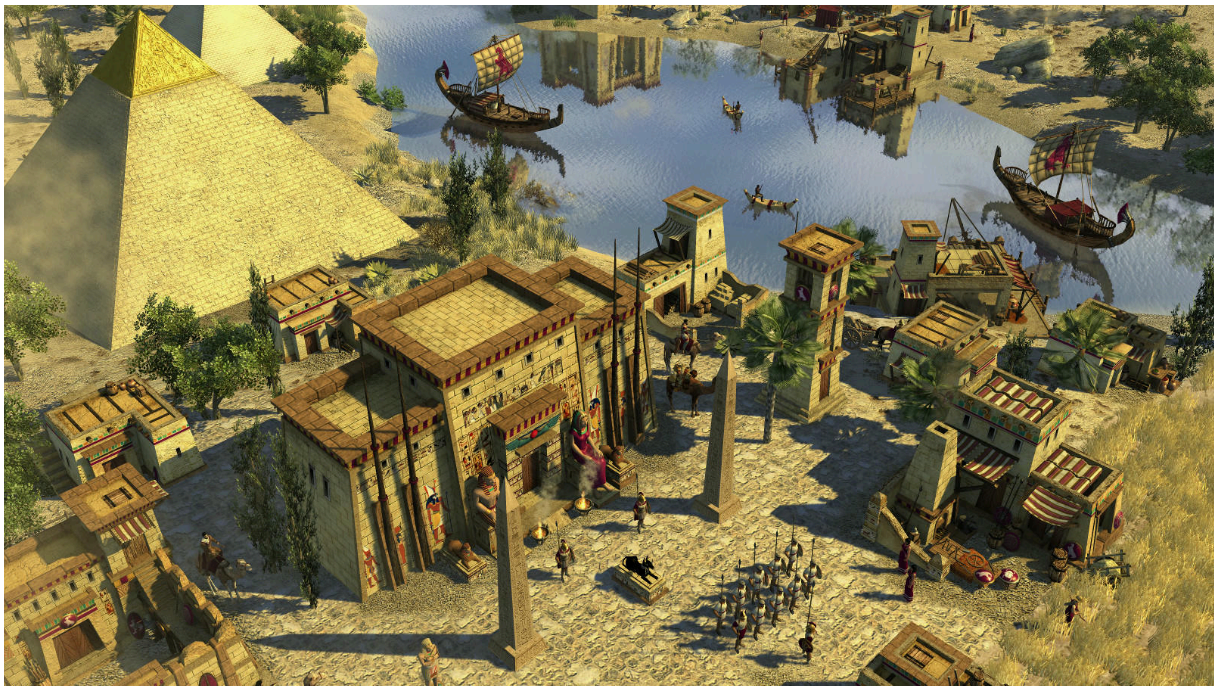


Figure 10: A screenshot from 0.A.D. showing the 3D assets in action.

3.4.7 ambientCG

ambientCG is a collection of public domain high-resolution physically-based materials created by Lennart Demes. It also contains a few 3D models with textures.

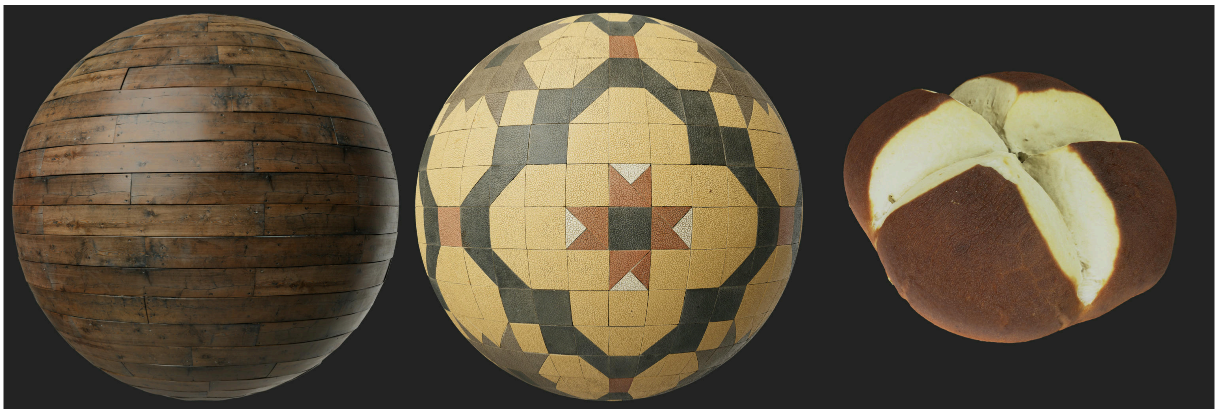


Figure 11: 2 materials and a model from ambientCG

[3]See files under `binaries/data/mods/*/art`

3.4.8 MakeHuman

To quickly create human character models with bones, you can use MakeHuman. The easiest way is with its blender plugin.

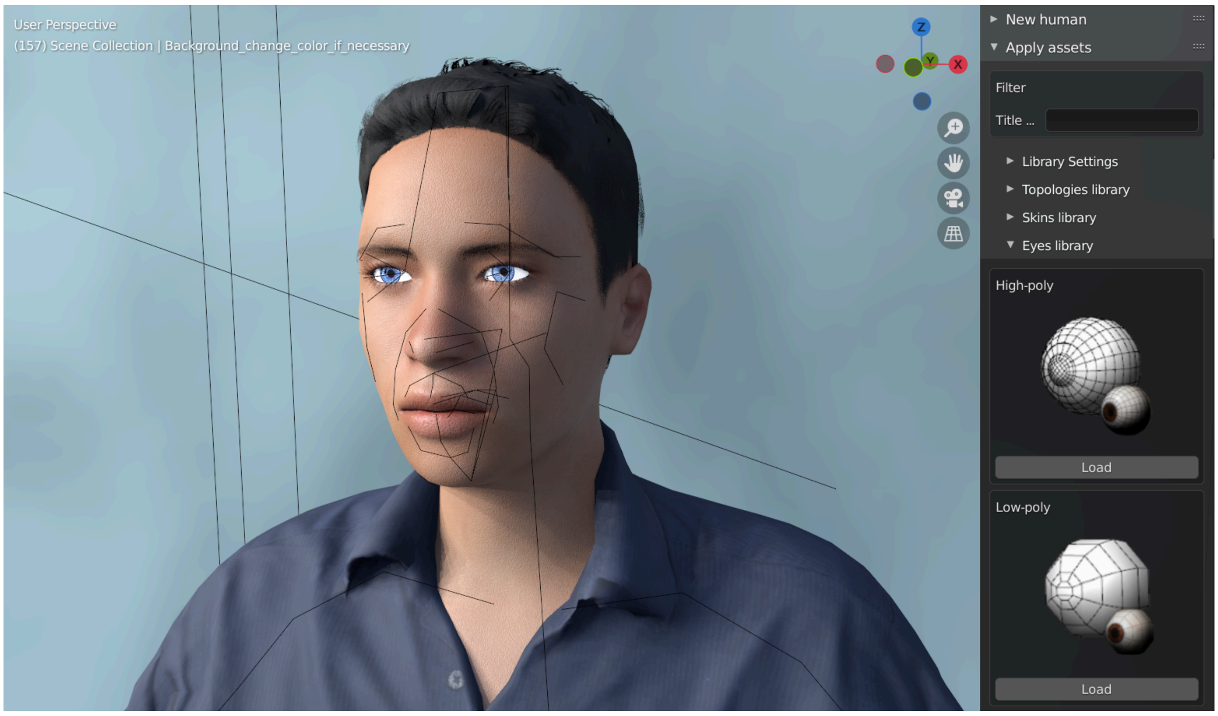


Figure 12: Screenshot of MakeHuman's blender plugin from their docs.

4 How to Godot

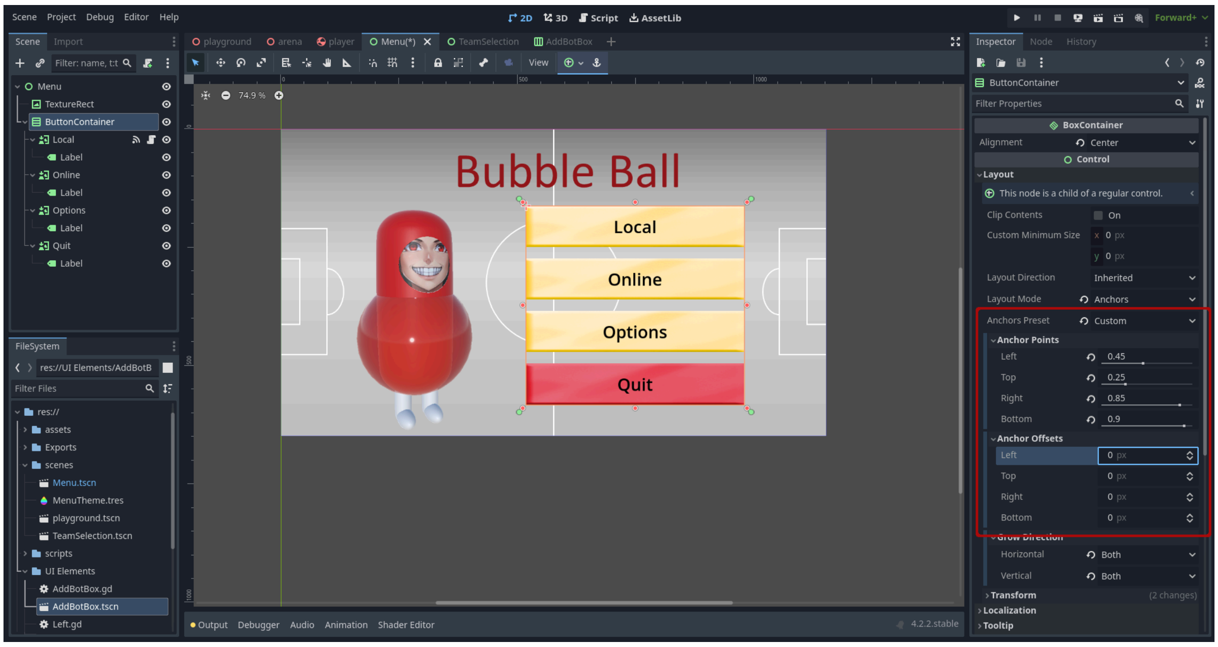
Godot is an open-source game engine under the MIT license. It uses an object-oriented programming model and has its own scripting language GDScript with optional typing and python-like syntax. It also supports C#/.NET and has native bindings called GDExtensions for C and C++. There are also several community-maintained language bindings like Kotlin or JavaScript/TypeScript with a modified base engine or Rust, Go, and Swift for GDExtensions. However, I advise using GDScript or C# as they have the best support, and have a programming model fitting the engine.

Some general hints for Godot:

- All fields accepting numbers also accept math expressions like `sqrt(2) - 1`.
- You navigate in the 3D panel, by pressing the right mouse button and move around with WASD. Press `Shift` to fly faster.

4.1 UI

UI components all inherit from the `Control` class. As simple but powerful layout containers, I recommend using `BoxContainer`s or `FlowContainer`s, which work conceptually similar to flexbox in the HTML document model. `HBoxContainer` and `VBoxContainer` stretch elements along the cross axis and cannot wrap in main direction. `HFlowContainer` and `VFlowContainer` don't stretch elements along the cross axis and do wrap in main direction. Elements whose position and size is not determined by the parent container, can use either absolute positions or anchors.



ButtonContainer

BoxContainer
Alignment: Center

Control
Layout
This node is a child of a regular control.
Clip Contents: On
Custom Minimum Size: x 0 px, y 0 px
Layout Direction: Inherited
Layout Mode: Anchors
Anchors Preset: Custom
Anchor Points
Left: 0.45
Top: 0.25
Right: 0.85
Bottom: 0.9
Anchor Offsets
Left: 0 px
Top: 0 px
Right: 0 px
Bottom: 0 px
Grow Direction
Horizontal: Both
Vertical: Both
Transform (2 changes)
Localization
Tooltip

Figure 13: The `VBoxContainer` named `ButtonContainer` in this example is anchored to be in a rectangle with $0.45 \cdot$ `width` as its left border. The anchor point rectangle is shown by the green pins in the main view. The rectangle after adding the absolute anchor offsets to each border is shown as orange box in the main view.

Text colors of `Label`s, space between of container elements, and similar properties are defined by the theme and can be overwritten by individual elements. To theme your UI, set the theme in the topmost UI node.

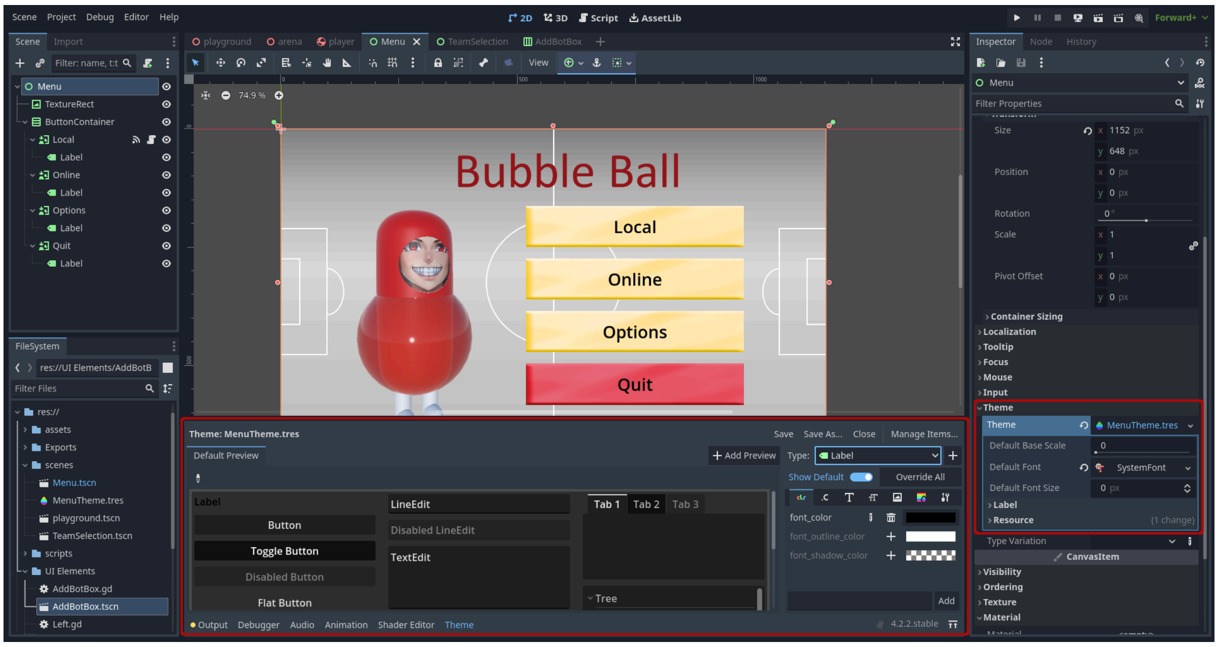


Figure 14: In the right box, you set the theme for a node and all it's children. On the bottom panel, you have a preview on the left and a panel to edit the properties (f.e. the text color) of UI types on the right.

4.2 Input processing

Godot can handle keyboard, mouse, touchscreen, controllers, joystick, and gamepad inputs. Controllers, joysticks or gamepads are modelled the same and named *Joypads* in the Godot docs. Touch devices are processed as mouse events with the `device` property set to -1.

There are two ways to process inputs. One method is, to handle inputs by setting actions in `Projects→Project Settings...→Input Map (Tab)`, and then question in `_process` whether `is_action_pressed` or `is_action_just_pressed` on the `Input` singleton is true. If you have multiple input devices the action map does not distinguish between them, so instead you have to check for keys with `is_key_label_pressed` or `is_physical_key_pressed`, or joypad inputs with `get_joy_axis` or `is_joy_button_pressed`.

To know what gamepads are connected you can either query for them with `get_connected_joypads` or observe the `joy_connection_changed` signal.

As a second method, instead of querying input state, you can also process input events as shown in the Godot docs.

To change a rigid body's velocity directly (not via impulse or force) based on user input, override `state.linear_velocity` in `_integrate_forces`, don't change physics state in `_process` or `_physics_process`.

4.3 Physics

You can generate a collider from a `MeshInstance3D` by selecting it, then open the `Mesh` options as shown in Figure 15.

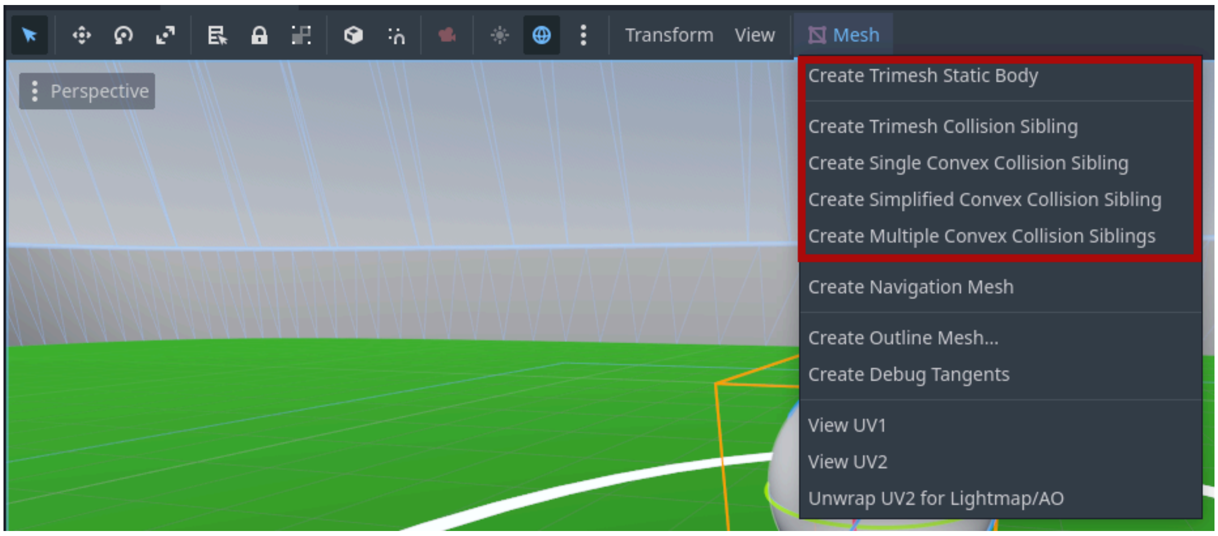

Transform
View
Mesh
Perspective
Create Trimesh Static Body
Create Trimesh Collision Sibling
Create Single Convex Collision Sibling
Create Simplified Convex Collision Sibling
Create Multiple Convex Collision Siblings
Create Navigation Mesh
Create Outline Mesh...
Create Debug Tangents
View UV1
View UV2
Unwrap UV2 for Lightmap/AO

Figure 15: "`Create Trimesh Static Body`" creates a static body (position and rotation are locked) with a complex collider. The other options create dynamic rigid bodies with colliders of varying complexity for this mesh.

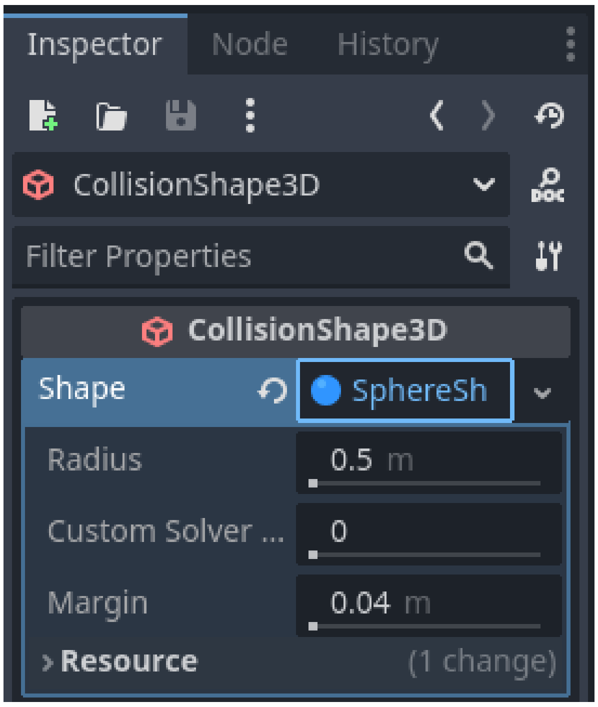


Figure 16: Inspector view, with a simple shape (here a sphere) selected

Instead of generating colliders, you can also add a child of type `CollisionShape3D` and select a simple shape in its shape attribute. This is preferable performance-wise.

4.4 Networking

See [the docs] for an introduction to Godot's high-level networking and sample code to build a client-server based lobby. The `multiplayer` variable is global, and assigning a client or server peer object to it enables multiplayer via the high-level API.

If you want spawned scenes to replicate on spawn across the network, you will need `MultiplayerSpawner`. To sync state continuously, you will need a `MultiplayerSynchronizer` child for the synchronized node, which has a script, that sets the properties, that should be synced, in the `replication_config`.

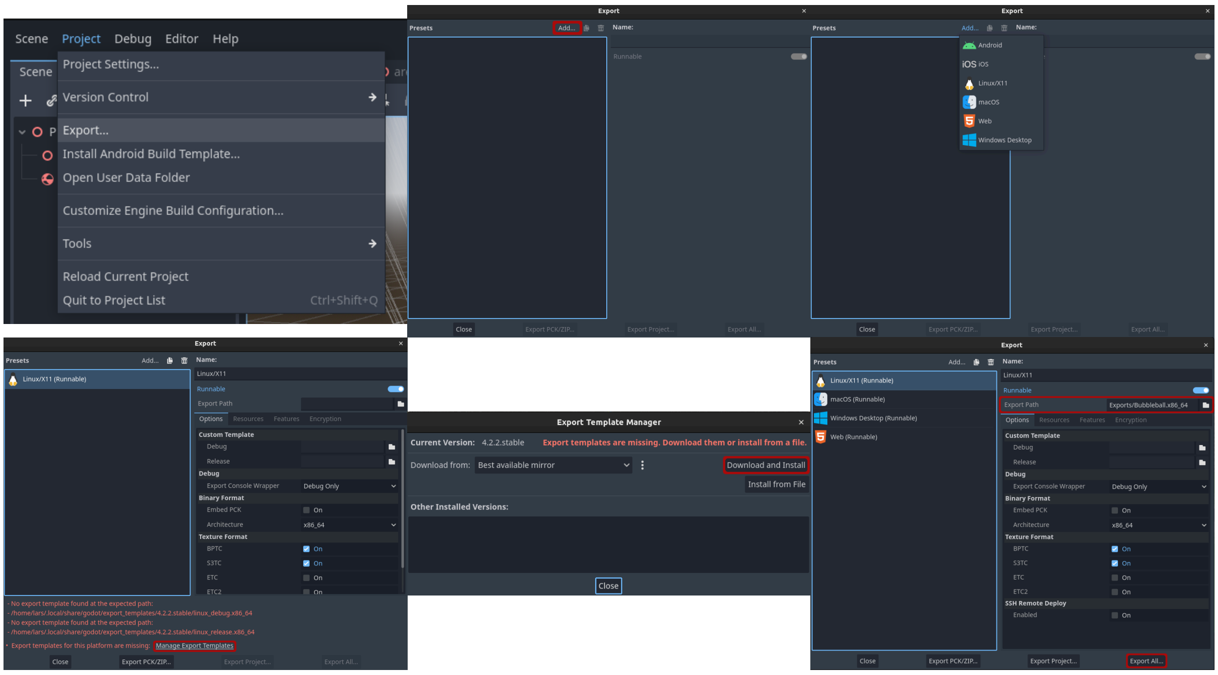
If you sync the positions of rigid bodies, you should `freeze` them in `FREEZE_MODE_KINEMATIC` on peers that don't control the body.

4.5 Global state / Keep state across scenes

https://docs.godotengine.org/en/stable/tutorials/scripting/singletons_autoload.html

4.6 How to cross-compile?

To export Godot games to different platforms, you can read their manual, or follow Figure 17.



- No export template found at the expected path:
- /home/lars/.local/share/godot/export_templates/4.2.2.stable/linux_debug.x86_64
- No export template found at the expected path:
- /home/lars/.local/share/godot/export_templates/4.2.2.stable/linux_release.x86_64

• Export templates for this platform are missing: Manage Export Templates

Current Version: 4.2.2.stable Export templates are missing. Download them or install from a file.

Figure 17: Open `Project→Export` for the export menu. There you can add targets with `Add`. After adding the first target, Godot will complain that you have no export templates. By click on `Manage Export Templates` you can open the dialog to download them. After downloading the templates, you can add all targets and set the export path for each target. You can either export a single target with `Export Project` or all with `Export All...` after configuring the export path for every target.

To use headless mode, simply add the `--headless` option when executing the game.

To run on macOS, enroll the apple developer program or disable gatekeeper.

5 How To Unity

Unity is a proprietary engine that might be used at no cost for small projects (see the EULA and whatever you have to consent, when downloading a license key). It only officially supports C# as scripting language. Other .NET language might work, but I discourage using them, and recommend sticking to the only language that is well tested, where bugs are fixed, and that is used in all Unity documentation.

To obtain Unity, use UnityHub. You will also need to register as student or personal user.

Using Git with Unity is generally a nightmare, because Unity uses UUIDs in its files, making them difficult to read, and randomly changes some fractional digits when different systems load a file and save it unchanged. Use this gitignore as a starting point, to get some control over the giant pile of files Unity will create.

5.1 UI

The GalacticKittens sample project referenced below still uses the old UI system. It is simple to start with, but inflexible and unresponsive. The new system UI Toolkit builds UIs from XML and CSS with flexbox for layouts.

```xml
<ui:UXML
  xmlns:ui="UnityEngine.UIElements" xmlns:uie="UnityEditor.UIElements"
  xsi="http://www.w3.org/2001/XMLSchema-instance"
  engine="UnityEngine.UIElements" editor="UnityEditor.UIElements"
  noNamespaceSchemaLocation="../UIElementsSchema/UIElements.xsd"
  editor-extension-mode="False"
>
  <ui:Style src="OverlayBig.uss"/>
  <ui:VisualElement name="full-screen-column">
      <ui:VisualElement name="top-row">
          <ui:Label class="fixed-portion" text="Testgame" name="mode"/>
          <ui:Label class="fixed-portion" text="Left" name="team-left"/>
          <ui:Label class="fixed-portion" text="" name="score"/>
          <ui:Label class="fixed-portion" text="Right" name="team-right"/>
          <ui:Label class="fixed-portion" text="0:00" name="time"/>
      </ui:VisualElement>
  </ui:VisualElement>
</ui:UXML>
```

Listing 1: UXML example

```css
#top-row {
	flex-grow: initial;
	flex-direction: row;
	justify-content: center;
	font-size: 36;
	-unity-font: url("../Fonts/OpenSans-Bold.ttf");
	background-color: lightgray;
	height: 5%;
	/* vertical horizontal */
	padding: 2px 8px;
	margin: 0;
}
```

Listing 2: USS example

5.2 Input processing

There are two input systems. The old one, called Input Manager, and the new one in the InputSystem-Package. The new input system has a simple way (actionmaps) and a complex way (event handling) of processing user input.

The simple way is shown by the docs. You create an `*.inputactions` file with actions that are then mapped to `On<actionname>` methods. This works if you have only one input device per local game instance. However, it is unsuited if you want to allow a multiplayer mode by connecting multiple gamepads to the same PC. In that case you need the to handle InputEvents that know what device triggered them.

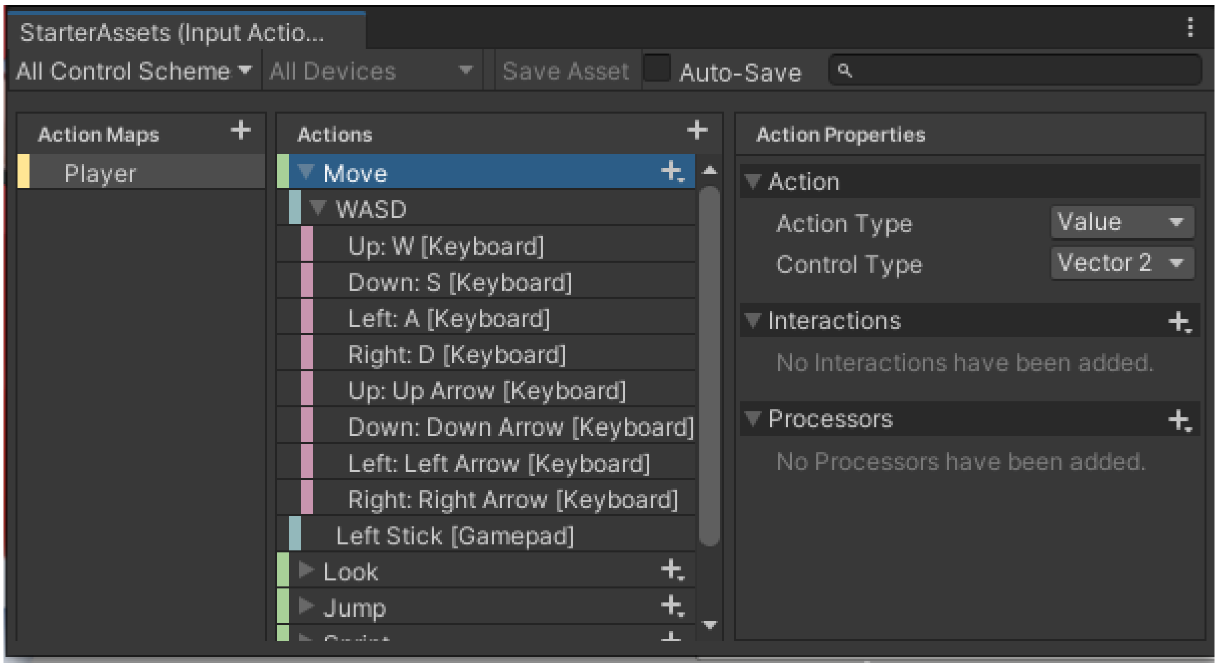
The complex way, used in that case, is to register handlers for events in `UnityEngine.InputSystem.InputSystem`.

```cs
// on device connect or disconnect
void OnDeviceChange(InputDevice device, InputDeviceChange change);
// on input
void OnInputEvent(InputEventPtr inputEvent, InputDevice inputDevice);

// register
UnityEngine.InputSystem.InputSystem.onDeviceChange += OnDeviceChange;
UnityEngine.InputSystem.InputSystem.onEvent += OnInputEvent;

// unregister
UnityEngine.InputSystem.InputSystem.onDeviceChange -= OnDeviceChange;
UnityEngine.InputSystem.InputSystem.onEvent -= OnInputEvent;
```

Listing 3: Usage of InputSystem Package



Player

Move
WASD
Up: W [Keyboard]
Down: S [Keyboard]
Left: A [Keyboard]
Right: D [Keyboard]
Up: Up Arrow [Keyboard]
Down: Down Arrow [Keyboard]
Left: Left Arrow [Keyboard]
Right: Right Arrow [Keyboard]
Left Stick [Gamepad]
Look
Jump

Action
Action Type: Value
Control Type: Vector 2
Interactions
No Interactions have been added.
Processors
No Processors have been added.

Figure 18: Example of an action map with an action `Move` of type 2D vector, that can be triggered on keyboard by WASD keys or with on gamepad with the left stick.

5.3 Networking

Use NetCode GameObjects. You can see it in use in GalacticKittens. See especially the `NetworkManager` component in the `Scenes/Bootstrap`, which has a list of Prefabs that need synchronization over the network. See also the `Network Object`, `Client Network Transform`, and `Network RigidBody` components in `Prefabs/Net/Player/PlayerShipBase`, which are needed to determine what properties should be synchronized.

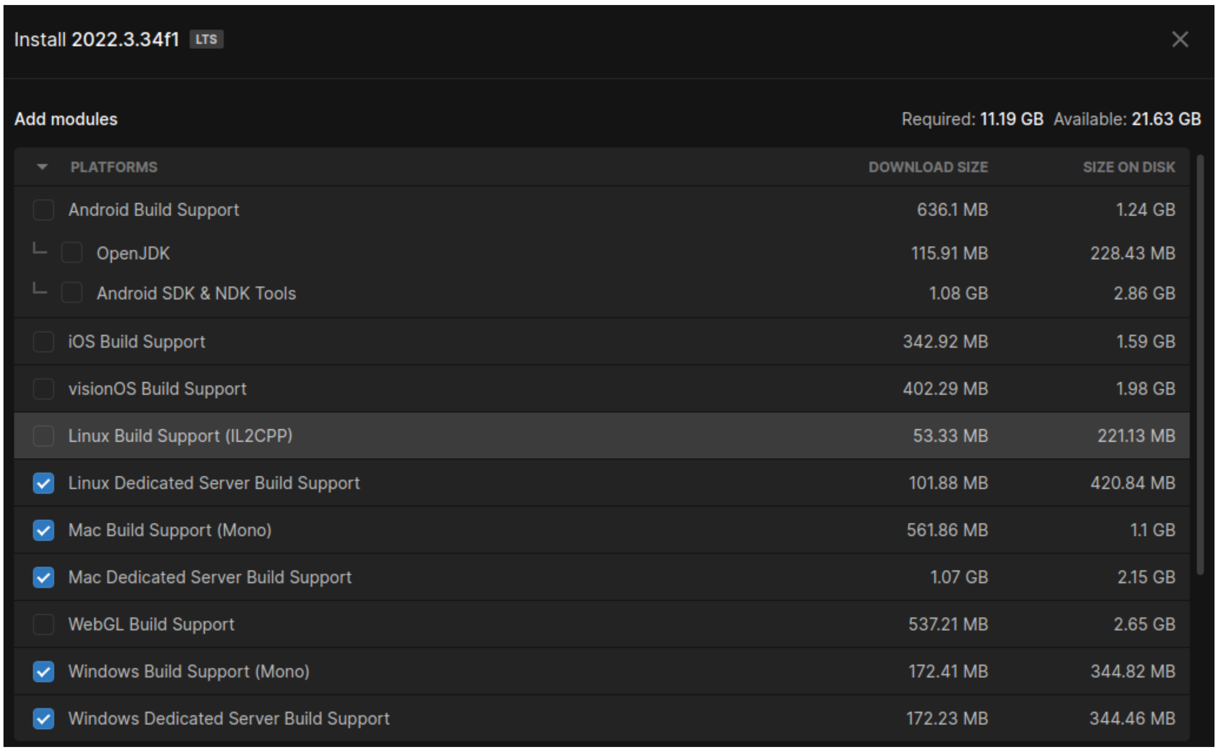
5.4 Global state / Keep state across scenes

Keep entities with `DontDestroyOnLoad` after scene and initialize them in a bootstrapping scene. Again, see GalacticKittens.

5.5 How to cross-compile?

You can adapt the build file from my bubbleball project and put it into `Assets/Editor/Builds.cs` to create a Unity Toolbar Item for builds.

To make it work, you need to have the modules for all supported platforms installed via UnityHub as seen in Figure 19.


Install 2022.3.34f1 LTS

Add modules — Required: 11.19 GB Available: 21.63 GB

PLATFORMS	DOWNLOAD SIZE	SIZE ON DISK
☐ Android Build Support	636.1 MB	1.24 GB
└ ☐ OpenJDK	115.91 MB	228.43 MB
└ ☐ Android SDK & NDK Tools	1.08 GB	2.86 GB
☐ iOS Build Support	342.92 MB	1.59 GB
☐ visionOS Build Support	402.29 MB	1.98 GB
☐ Linux Build Support (IL2CPP)	53.33 MB	221.13 MB
✓ Linux Dedicated Server Build Support	101.88 MB	420.84 MB
✓ Mac Build Support (Mono)	561.86 MB	1.1 GB
✓ Mac Dedicated Server Build Support	1.07 GB	2.15 GB
☐ WebGL Build Support	537.21 MB	2.65 GB
✓ Windows Build Support (Mono)	172.41 MB	344.82 MB
✓ Windows Dedicated Server Build Support	172.23 MB	344.46 MB

Figure 19: Module options on Linux for Linux, macOS, and Windows support for both graphical and headless (dedicated server) mode. Since this screenshot was done on Linux, there is no `Linux Build Support (Mono)` option, which is built-in on Linux. If you are on Windows, the `Windows Build Support (Mono)` option would be built-in and `Linux Build Support (Mono)` has to be selected additionally.

6 Copyright notice