Limon Engine Documentation

Release 0.6

Limon Engine

Apr 01, 2019

Contents

CHAPTER 1

Welcome to Limon Engine's documentation!

Limon is a multi platform 3D game engine mainly focusing on first person games. Focus of its development is ease of use and ease of study. It has builtin editor for maps, animations and GUIs.

Prebuilt binaries for Windows, Linux and MacOS can be found here

If you want to check out what Limon engine is and can do, you can check out about.

The documentation is separated in three major sections:

1. How to use the engine and build in editor to create games.
2. Extending capabilities using Limon API.
3. Engine architecture and internals.

1.1 Contribute

If you are having issues, or if you think some awesome feature is missing, please let us know using the issue tracker. Also if you want to chat, there is a Discord channel.

- Issue Tracker: https://github.com/enginmanap/limonEngine/issues
- Source Code: https://github.com/enginmanap/limonEngine
- Discord: https://discord.gg/gqprbFd

1.2 License

The project is licensed under the LGPL license.

1.2.1 Contents:

Getting Started

Acquiring Limon Engine

You can download the prebuilt binaries from github releases, or you can build the engine for yourself.

The engine comes with a test map, and launches it by default, so you can launch the engine and start working. If you are not interested in building on your own, you can continue to *Using Builtin Editor*.

Building

Limon uses c++14, so a supporting c++ compiler is required. GCC 7.2 to 8.2 are tested. Building Limon also requires the libraries listed below:

- assimp
- bullet
- sdl2
- sdl2-image
- freetype (likely freetype6 as library name)
- tinyxml2
- glew
- glm

If you are using Ubuntu, you can use the line below to install the required libraries:

```
$ sudo apt install cmake git git-lfs libassimp-dev libbullet-dev libsdl2-dev libsdl2-
↪image-dev libfreetype6-dev libtinyxml2-dev libglew-dev build-essential libglm-dev␣
↪libtinyxml2-dev
```

Limon engine GitHub repository is configured to keep sample model files in git-lfs. If you don't have git-lfs installed, engine will compile as expected, and you can run your own maps with your own models, but sample map won't work. Following line can be used to clone the engine with git-lfs:

```
git lfs install
git clone https://github.com/enginmanap/limonEngine.git
cd limonEngine
git lfs pull
```

Limon Engine uses cmake as build system, if all the libraries are installed and cmake can find them, invoking cmake should build the engine.

In the cloned directory, call these commands:

```
mkdir build
cd build
cmake ../
cd ..
```

After cmake is done creating the build files, you can build and copy the sample data using these commands:

```
cd build
make
cp -a ../Data .
```

Running

Limon engine takes single parameter, and that is the path to first map file to load. If no parameter is passed, Limon first tries to get the file name from release settings file, at ./Data/Release.xml, if file not found, or no world name specified, it defaults to "./Data/Maps/World001.xml", which is a test map that has samples for capabilities of the engine.

The Custom trigger are automatically loaded from the same directory of the engine binary, with name *libcustomTriggers*, the extension of that file depends on the platform(dll, so, dynlib).

After engine launch, the key bindings are as follows:

- Pressing *0* switches to debug mode, renders physics collision meshes and disconnects player from physics (flying and passing trough objects)
- Pressing *F2* key switches to editor mode, which allows creating maps.
- Pressing + and - changes mouse sensitivity.
- *wasd* for walking around and mouse for looking around as usual.

The options of the game engine can be edited using ./Engine/Options.xml file.

Using Builtin Editor

Builtin editor can be used to create maps, interfaces and menus. 3D and 2D components share the tooling, and they can be mixed and matched, meaning it is possible to build 3d world as background in menus, or using menus with in game play.

Editor Basics

Editor mode uses a free cursor control scheme. You can move around by "WASD" and look around by moving mouse cursor to edges of the screen. Please take note, editor windows have precedence over these controls, so if mouse is over a control, or if a text input is selected, these controls wont work.

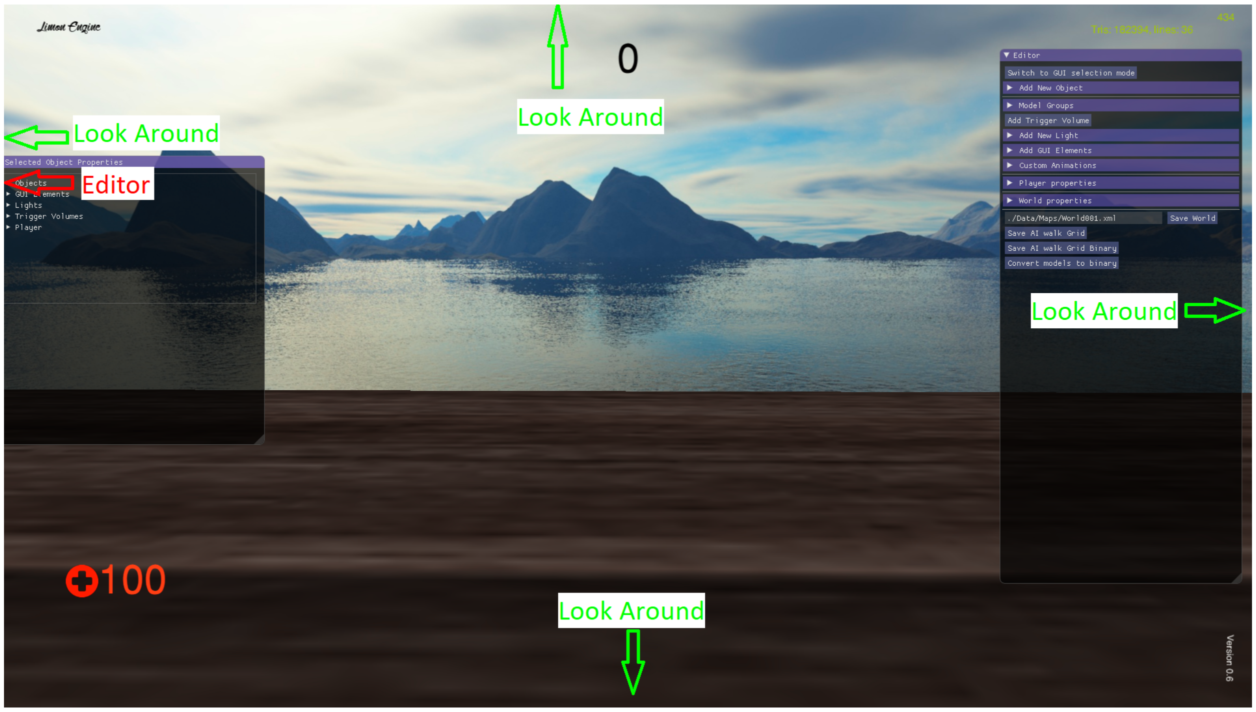
The editor has 3 main windows. By default left of the screen is object properties, right is world properties and bottom is animation sequencer. As expected you can drag and drop and resize the windows. If no object is selected, only the world properties window is visible. Animation sequencer is visible only when creating an animation.

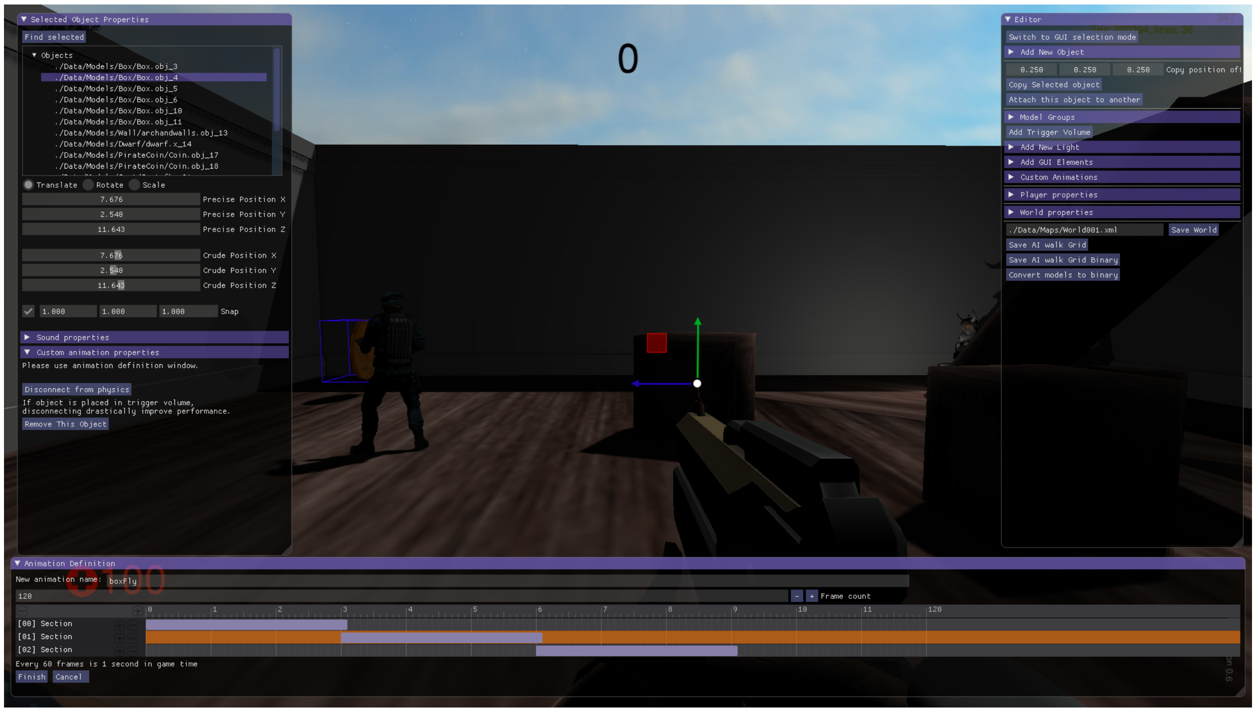
World Editor Details

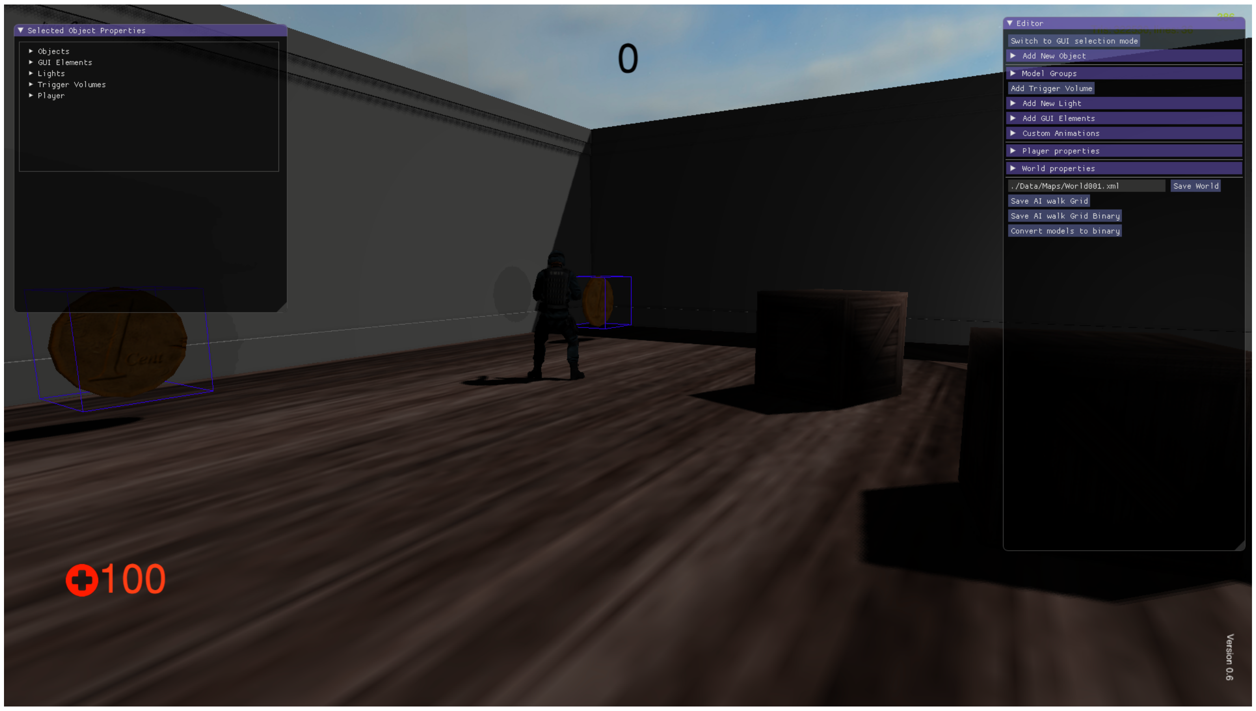
World editor loads up with almost all the features hidden by the titles. The fresh loaded window looks like this:

There are 2 main types of elements that can be used to build the world.

- 3D Objects
- Lights
- GUI elements


Limon Engine
0
Look Around
Look Around
Selected Object Properties
Objects
GUI Elements
Lights
Trigger Volumes
Player
Editor
Editor
Switch to GUI selection mode
Add New Object
Model Groups
Add Trigger Volume
Add New Light
Add GUI Elements
Custom Animations
Player properties
World properties
./Data/Maps/World001.xml
Save World
Save AI walk Grid
Save AI walk Grid Binary
Convert models to binary
Look Around
100
Look Around
Version 0.6


Selected Object Properties
Find selected
Objects
./Data/Models/Box/Box.obj_3
./Data/Models/Box/Box.obj_4
./Data/Models/Box/Box.obj_5
./Data/Models/Box/Box.obj_6
./Data/Models/Box/Box.obj_10
./Data/Models/Box/Box.obj_11
./Data/Models/Wall/archandwalls.obj_13
./Data/Models/Dwarf/dwarf.x_14
./Data/Models/PirateCoin/Coin.obj_17
./Data/Models/PirateCoin/Coin.obj_18
Translate
Rotate
Scale
7.676 Precise Position X
2.540 Precise Position Y
11.643 Precise Position Z
7.676 Crude Position X
2.540 Crude Position Y
11.643 Crude Position Z
1.000 1.000 1.000 Snap
Sound properties
Custom animation properties
Please use animation definition window.
Disconnect from physics
If object is placed in trigger volume, disconnecting drastically improve performance.
Remove This Object
0
Editor
Switch to GUI selection mode
Add New Object
0.250 0.250 0.250 Copy position of
Copy Selected object
Attach this object to another
Model Groups
Add Trigger Volume
Add New Light
Add GUI Elements
Custom Animations
Player properties
World properties
./Data/Maps/World001.xml
Save World
Save AI walk Grid
Save AI walk Grid Binary
Convert models to binary
Animation Definition
New animation name: boxFly
120
Frame count
0 1 2 3 4 5 6 7 8 9 10 11 120
[00] Section
[01] Section
[02] Section
Every 60 frames is 1 second in game time
Finish
Cancel



To make selection easier, click to select doesn't work for both types at the same time. At launch you can select Objects. If you click "Switch to GUI selection mode", you will be able to select GUI elements, but not Objects. The button will be replaced by "Switch to World selection mode".

Adding 3D objects

There are 2 types of 3D object you can add to the world.

- Models
- Trigger volumes

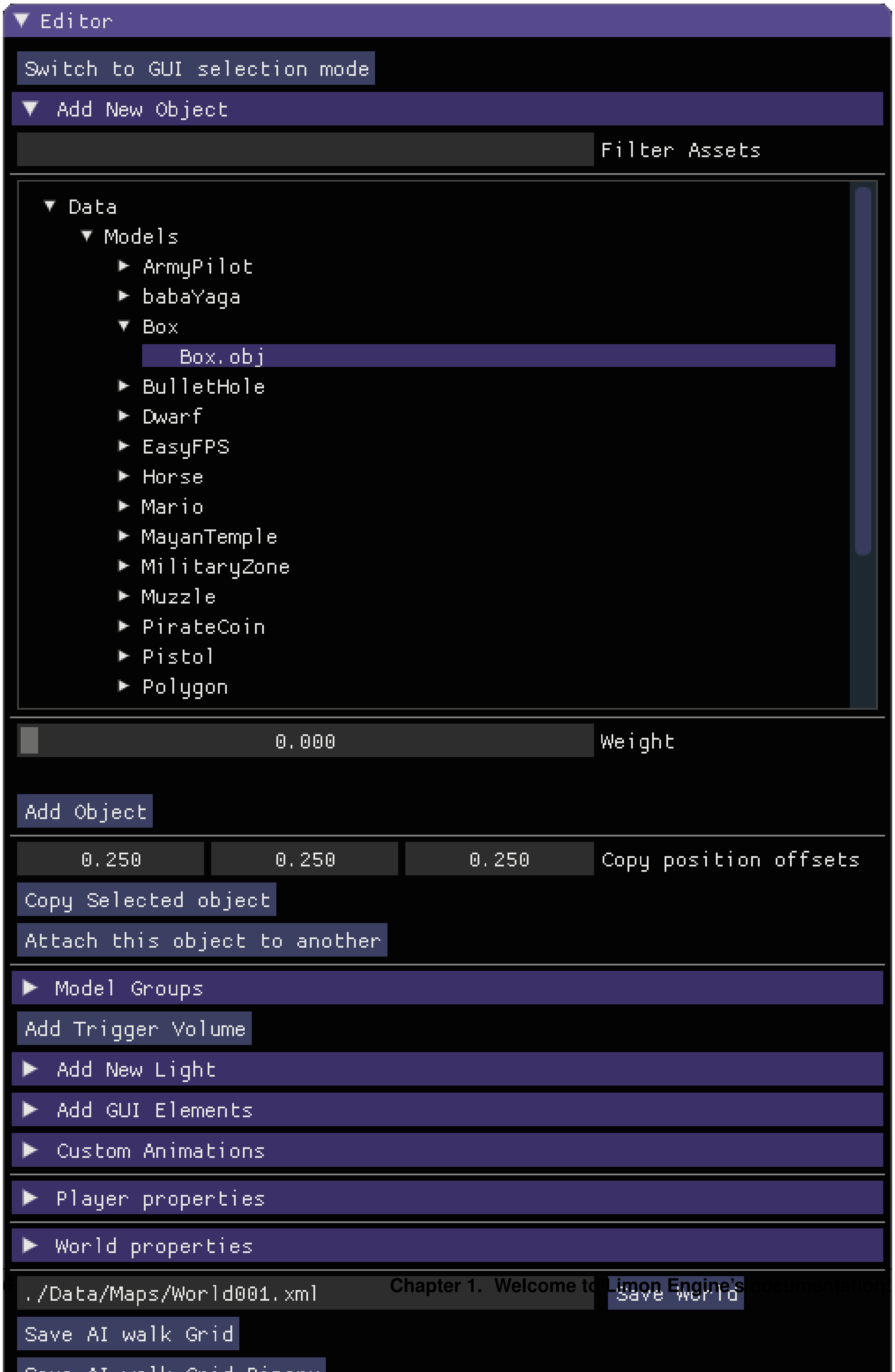
It is also possible to create a ModelGroup.

A directory tree of ./Data folder will be loaded, with directories that don't contain model files filtered out. For filtering based on file name, the text box just above can be used. When mass is set to 0, the object will be marked "static" by physics system. This means the model will not be able to move, outside of the editor. It is the default setting, to be used for building the world. Any other value will mark the object as "dynamic", meaning the objects movements will be governed by physics engine.

If an object is already selected, "Copy position offsets" settings and "Copy Selected Objects" button is shown. These are used to copy selected object, with given offsets. Offsets can be used to make copy automatically position so a grid or a wall can be easily created.

Dynamic objects physical representations are simplified automatically. For inanimate models, the simplified representation will be a convex hull. This means the cavities, crannies etc. will not be calculated for them. For animated models, each vertex will be assigned the bone that has the most weight on it, and for each bone a convex hull will be created. It means separate members of the model will be calculates as such.

Static object have a full mesh representing physical object. It is possible to replace this with a simplified mesh. To do so, bake meshes with names prefixed with "**UCX_**". They will be used for physics.


Editor
Switch to GUI selection mode
Add New Object
Filter Assets
Data
Models
ArmyPilot
babaYaga
Box
Box.obj
BulletHole
Dwarf
EasyFPS
Horse
Mario
MayanTemple
MilitaryZone
Muzzle
PirateCoin
Pistol
Polygon
0.000
Weight
Add Object
0.250
0.250
0.250
Copy position offsets
Copy Selected object
Attach this object to another
Model Groups
Add Trigger Volume
Add New Light
Add GUI Elements
Custom Animations
Player properties
World properties
./Data/Maps/World001.xml
Save World
Save AI walk Grid

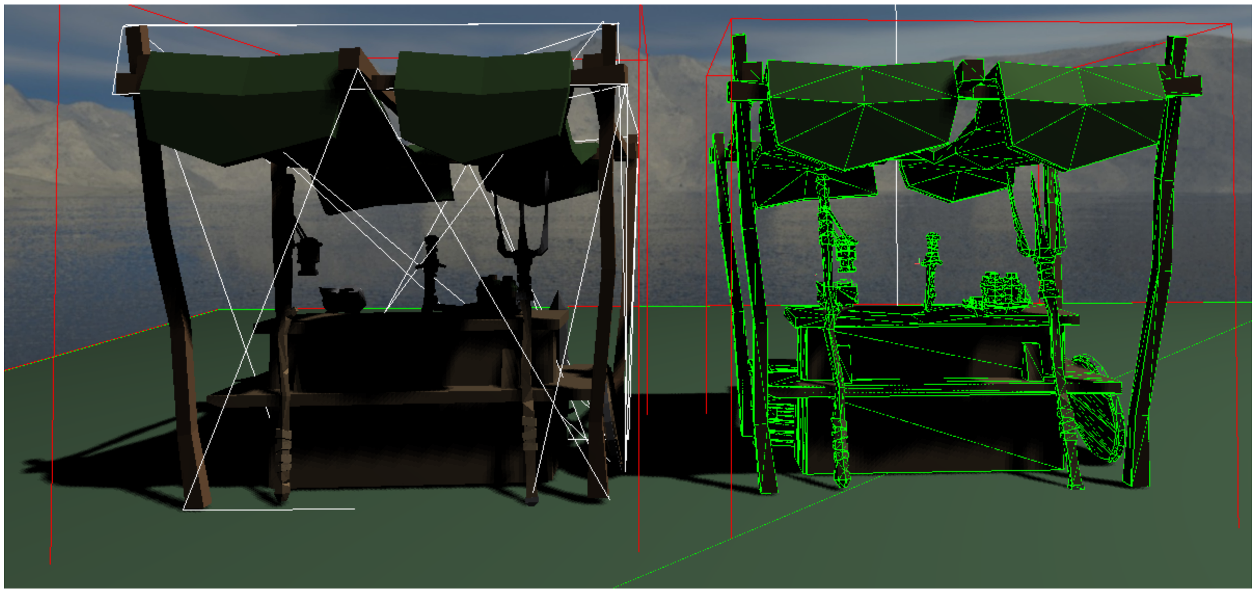


Fig. 1: Dynamic object have Convex hulls (left), while static objects have full mesh as collision mesh (right).

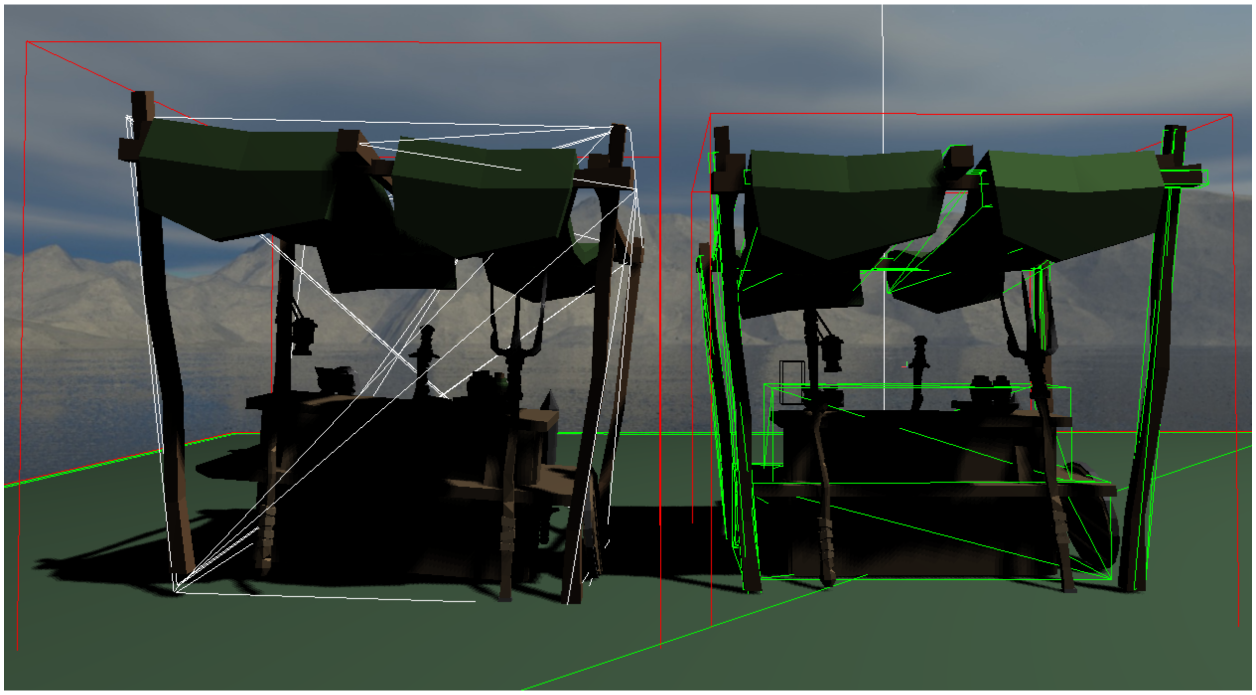


Fig. 2: Static objects can use baked collision meshes(right)

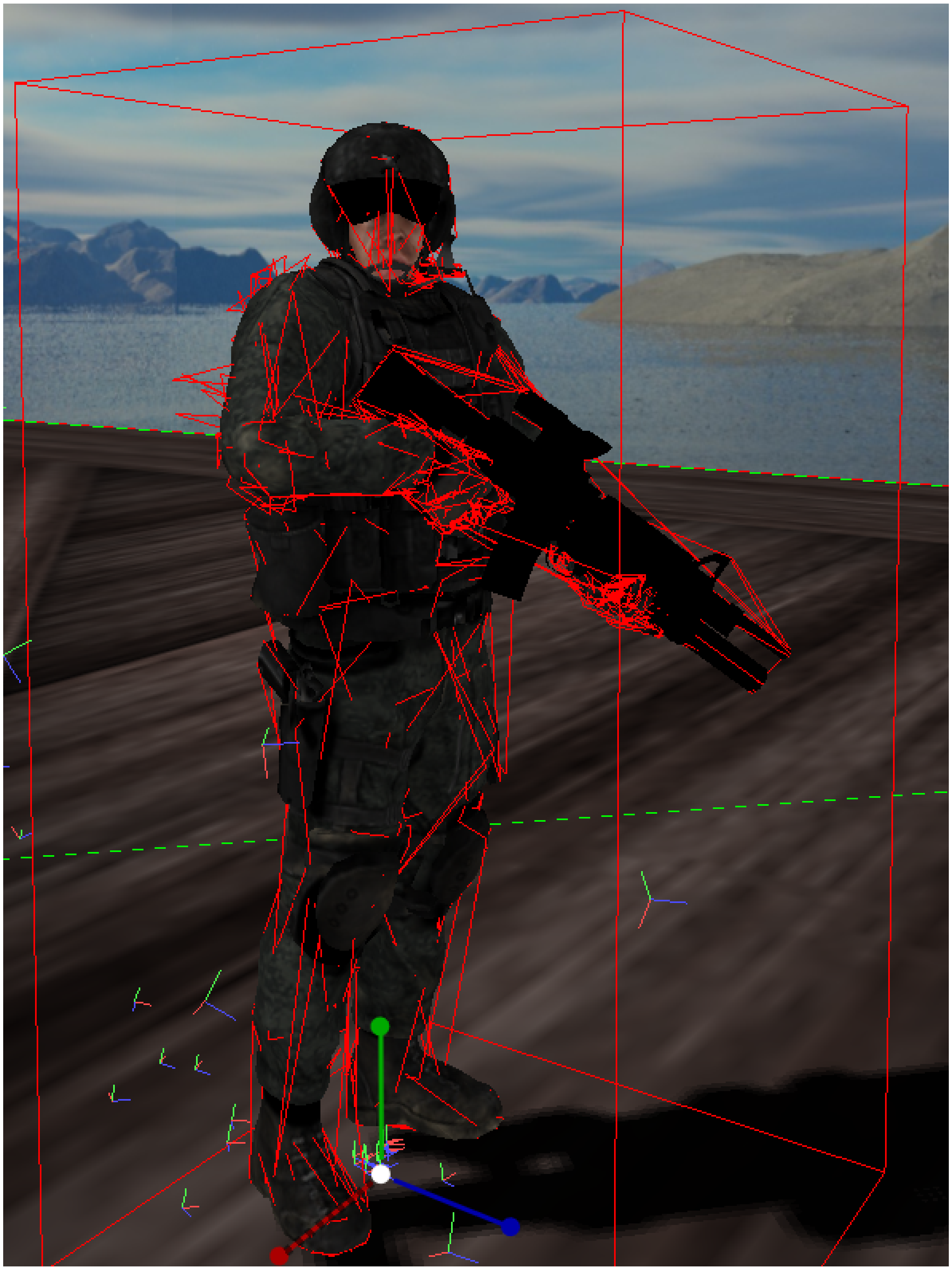


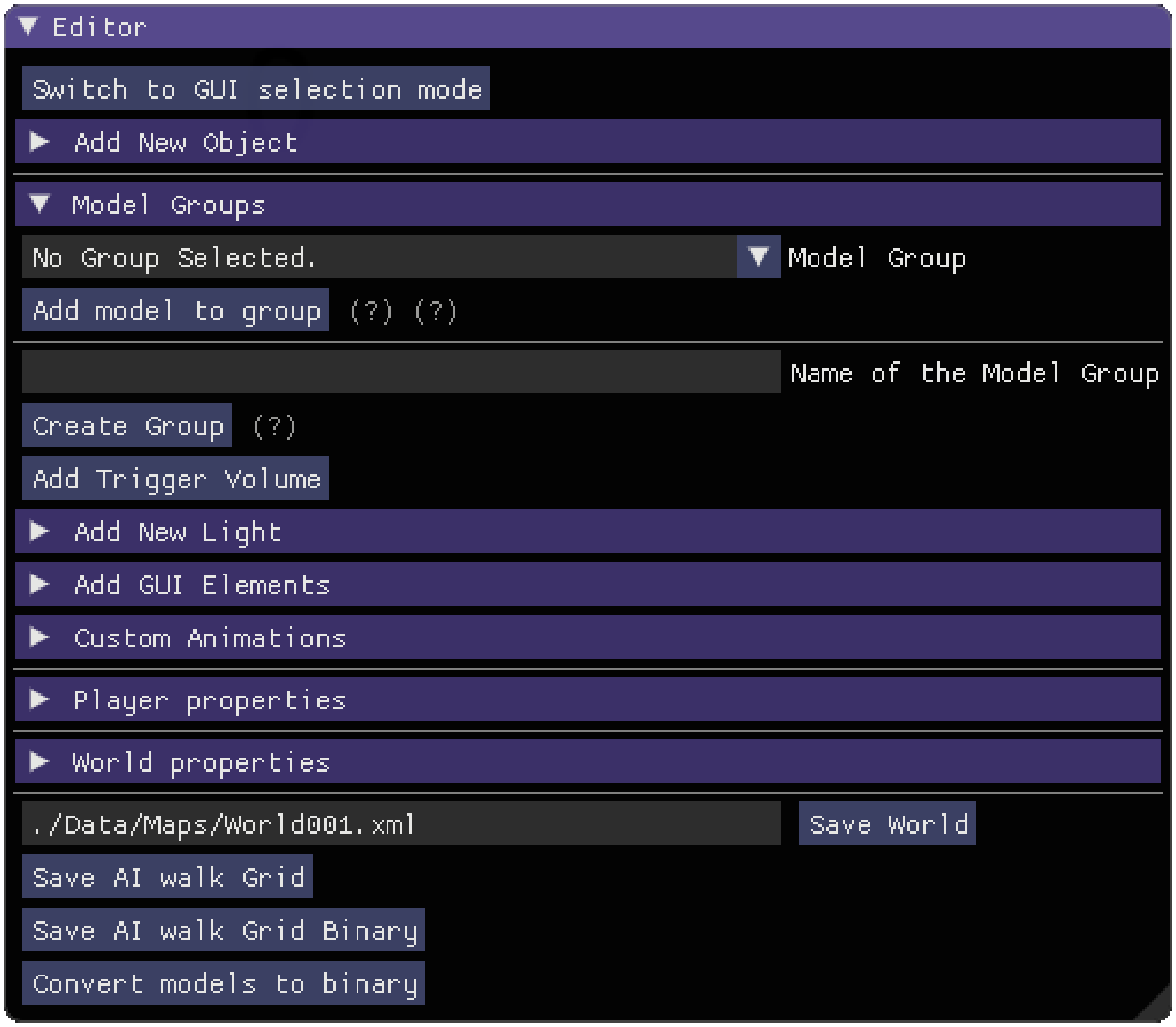
Fig. 3: animated object has per bone convex hulls

Note: Models with animations, both from the asset itself and custom using the editor are considered "kinematic" It means the object is allowed to move, but the movement is not governed by physics engine. Those types of objects can't be moved by physical interactions like pushing or pulling, but they can effect physical objects.

Note: Because physical representation is dependent on the mass, mass setting cant be changed once an model is added to the world. If you need to change the mass, remove and add again.

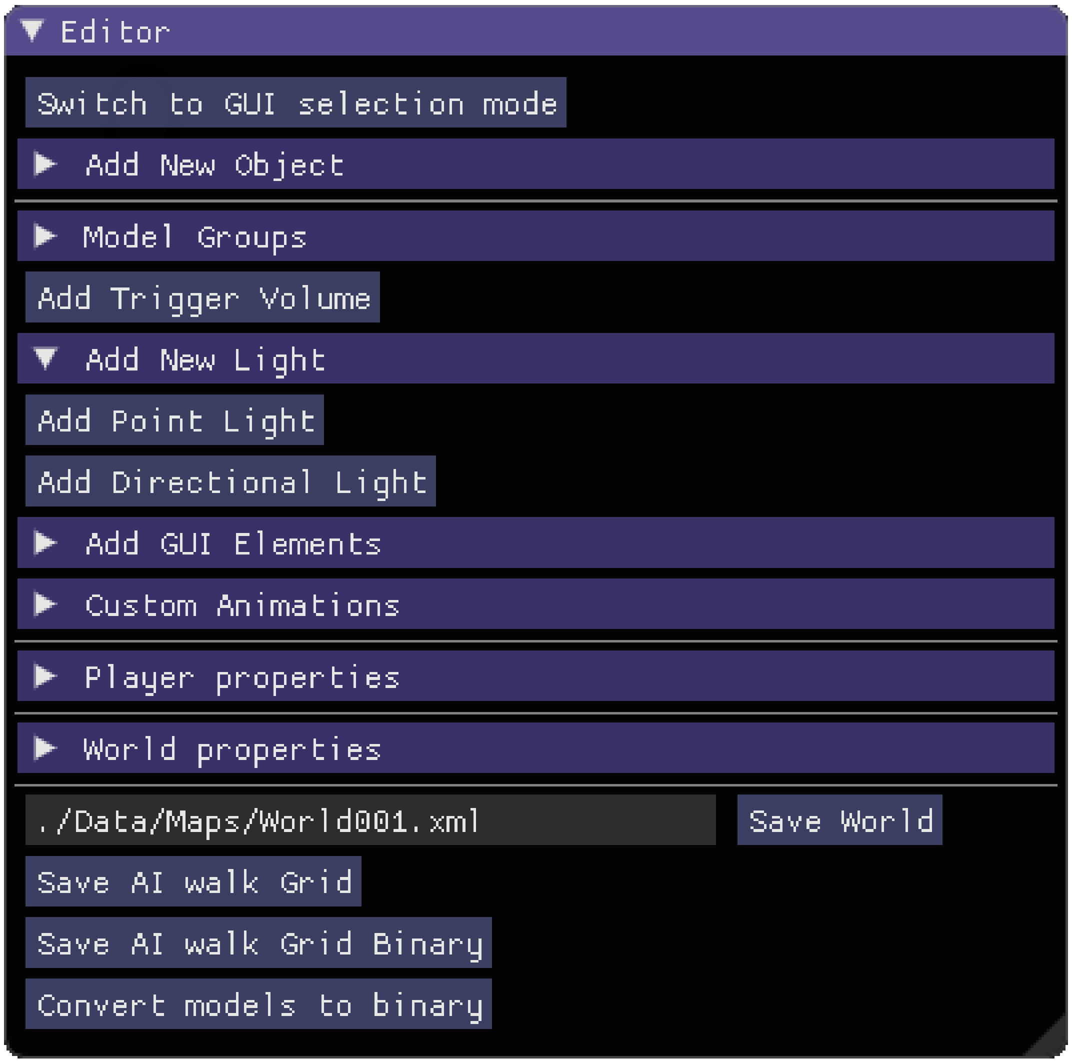
Add Trigger Volume button will create an empty cube. That cube can be used to trigger custom code paths. The details are at *Trigger Object Settings*.

Creating ModelGroups



A model group can hold unlimited number of models, or other model groups. It is used to make operations on multiple objects easier. First part of Model Groups section is used to put a model to a model group. Second part is used to create a new model group with given name.

Adding Lights



Limon Engine uses a custom forward renderer with full dynamic light calculation. Since these calculations are resource hungry, only 4 lights will be enabled at any given time. This value is set at compile time, and not expected to be changed by game developers. To make usage easier, only 1 directional light is allowed, but that light is never disabled (assumed Moon/Sun), so 1 directional + 3 point, or 4 point lights may be active at any given time. Engine itself decides which ones to activate and deactivate using player position, so adding more than 4 light is allowed.

Adding GUI Elements

GUI elements are rendered using layers. Each layer has a level, with default layer at level 0. Bigger levels are higher up, meaning when overlapped, the one with the higher level will be rendered.

You can add the following using the editor:

- Layer
- Text
- Image

- Button
- Animation

GUI Layer

Add GUI layer menu allows you to add a new layer, with set level.

GUI Text

To add GUI text, you need to set the font, font size and name. The layer of the text can be selected from the drop down.

Note: Text scaling will be converted to font size on next load, to provide better quality

GUI Image

When adding Image as GUI element, a directory tree of ./Data will be shown, filtered based on supported image formats. To filter based on file names, the fiter text box on top of directory listing can be used. The layer of the image can be selected from the drop down.

GUI Button

To add GUI Button, you need to set the name, and set normal image using directory tree. The rest of the fields are optional. For details please check *GUI Button Settings*. The layer of the button can be selected from the drop down.

GUI Animation

This Widget is not fully functional at 0.6 release. Please avoid until next release.

Player Properties

The Player properties section allows to set what is the launch time player mode. For game release, this should be either Menu, or Physical. Other types can be useful for development.

- Physical: Normal Player for game play
- Debug: The player that controls exactly like physical, but doesn't interact with physics, so can fly and walk-through objects. Also renders physics meshes, GUI borders and AI walk grid to allow debugging issues.
- Editor: Builtin editor.
- Menu: Menu interaction is allowed, movement and screen rotation disallowed. Mouse is set to free movement.

If a custom player extension is going to be used, entering its name will load and enable the extension. If player has a Model attached, there will be an "Disconnect Attachment" button.

Setting Up World Properties

The world properties is used to set map global properties.

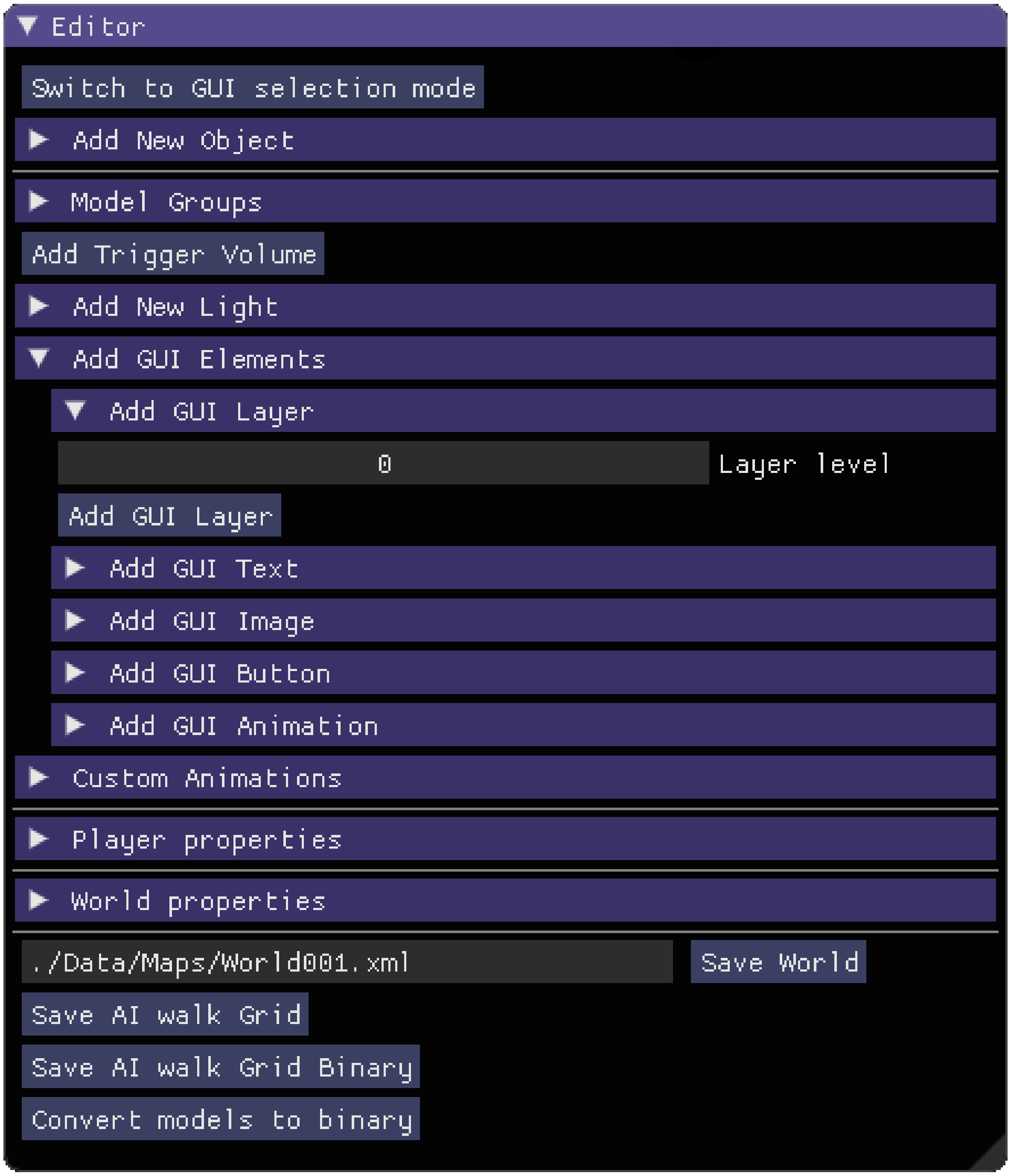
OnLoad Actions

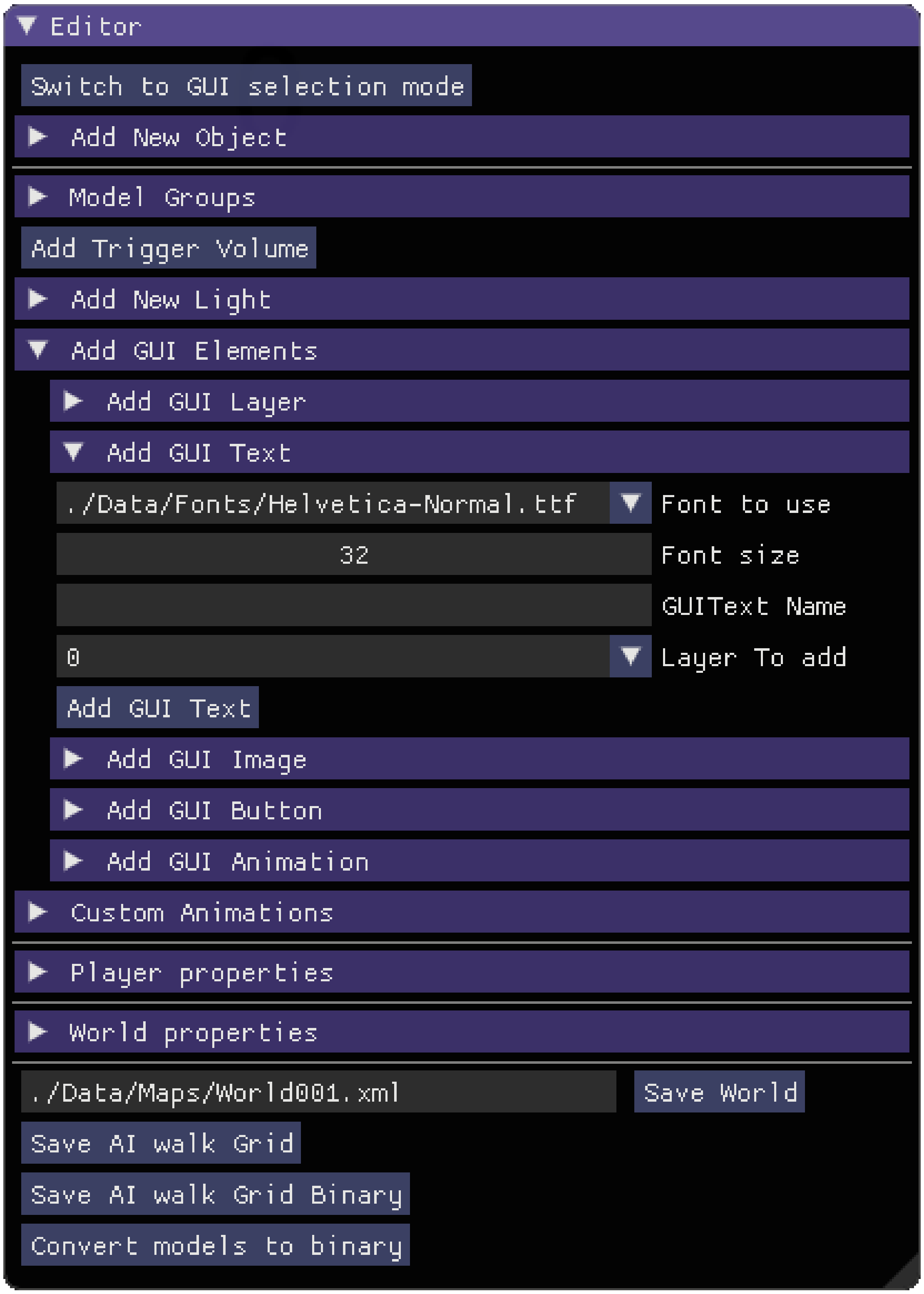
If map designer wants to launch some custom action at map load, this interface can be used to set as many as required. Details of them are at Triggers

Music

Music of the map can be set using the directory listing

SkyBox

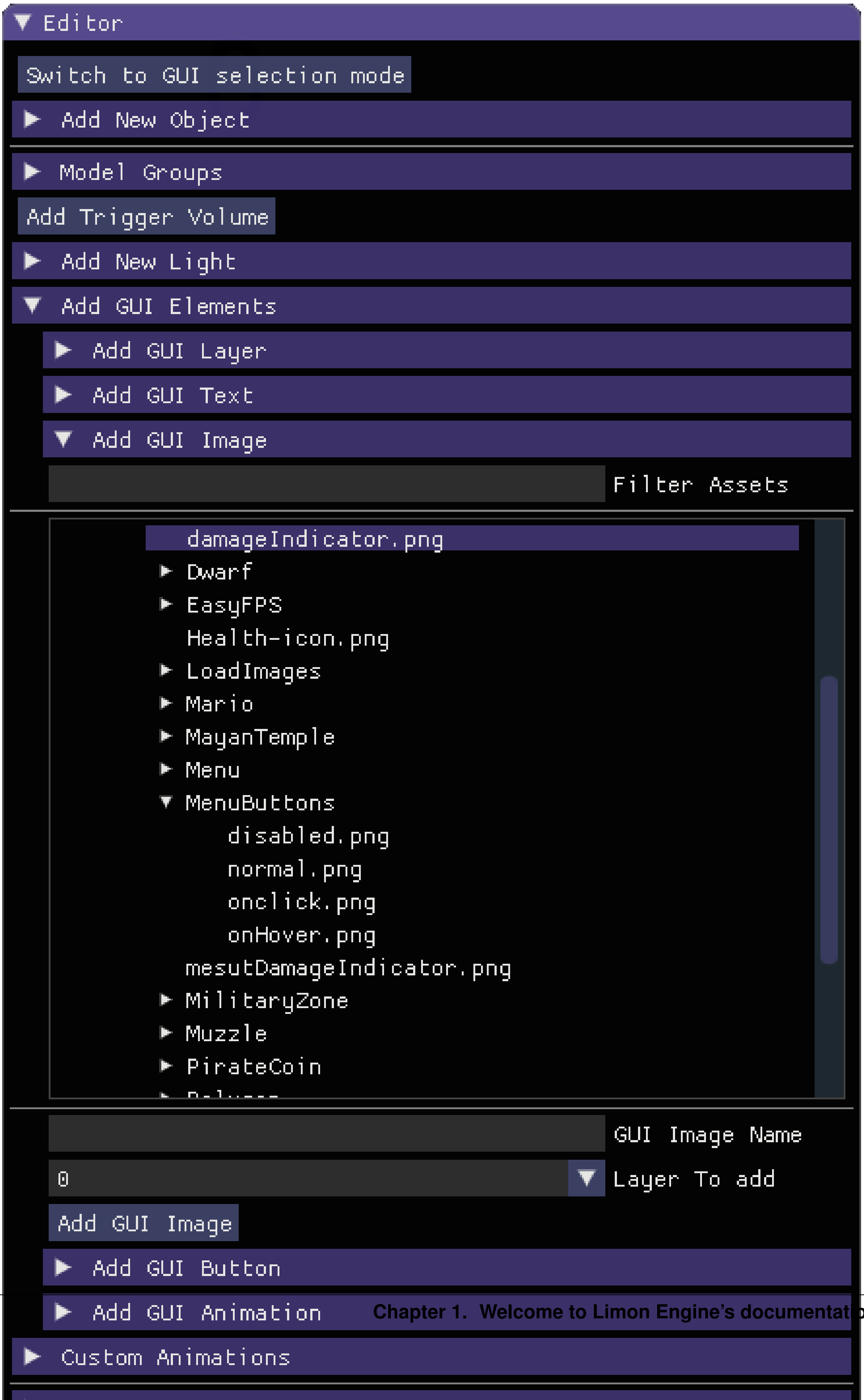

Editor
Switch to GUI selection mode
Add New Object
Model Groups
Add Trigger Volume
Add New Light
Add GUI Elements
Add GUI Layer
0
Layer level
Add GUI Layer
Add GUI Text
Add GUI Image
Add GUI Button
Add GUI Animation
Custom Animations
Player properties
World properties
./Data/Maps/World001.xml
Save World
Save AI walk Grid
Save AI walk Grid Binary
Convert models to binary

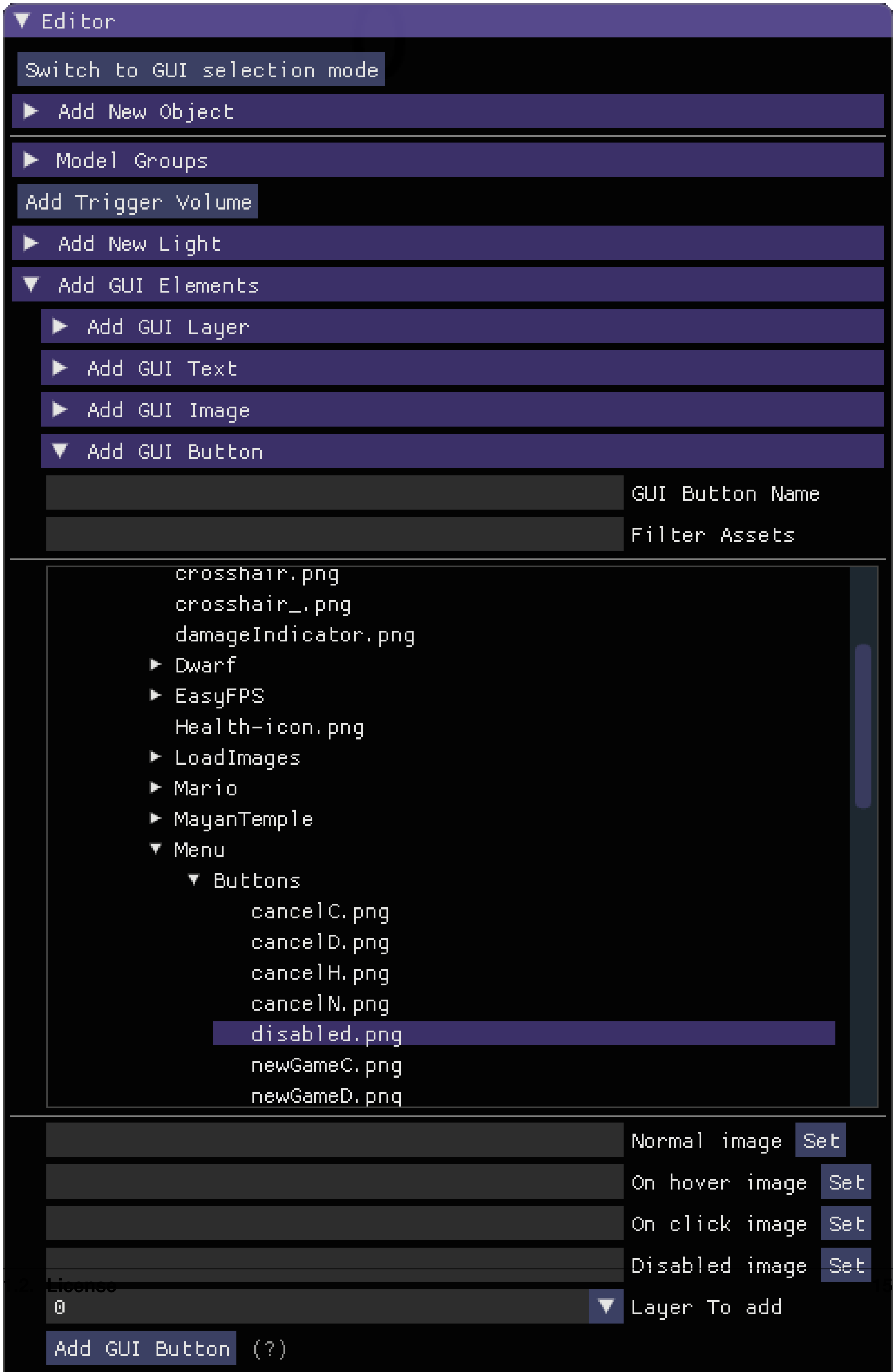


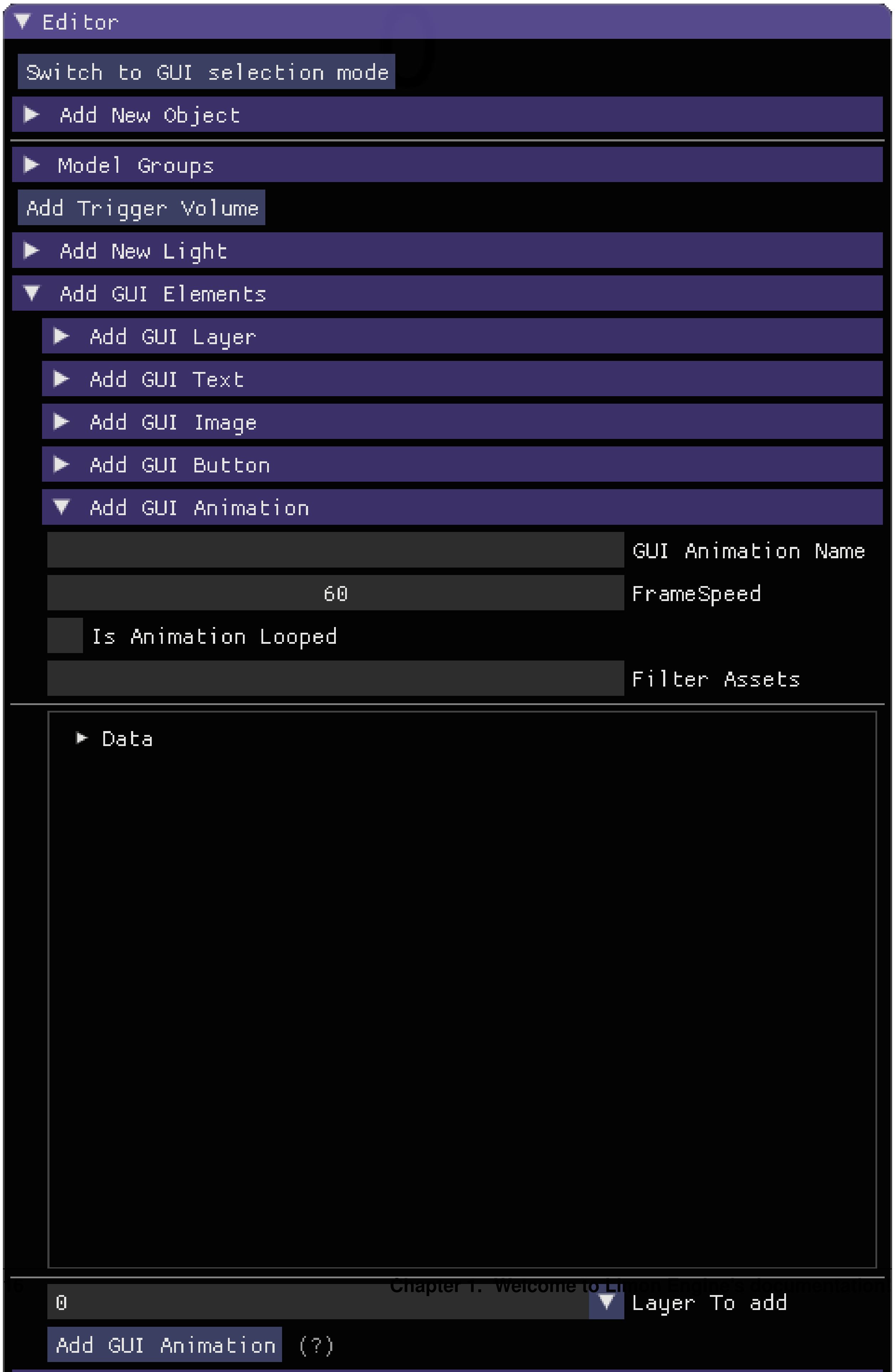
./Data/Fonts/Helvetica-Normal.ttf	Font to use
32	Font size
	GUIText Name
0	Layer To add

Add GUI Text

./Data/Maps/World001.xml


▼ Editor
Switch to GUI selection mode
▶ Add New Object
▶ Model Groups
Add Trigger Volume
▶ Add New Light
▼ Add GUI Elements
▶ Add GUI Layer
▶ Add GUI Text
▼ Add GUI Image
Filter Assets
damageIndicator.png
▶ Dwarf
▶ EasyFPS
Health-icon.png
▶ LoadImages
▶ Mario
▶ MayanTemple
▶ Menu
▼ MenuButtons
disabled.png
normal.png
onclick.png
onHover.png
mesutDamageIndicator.png
▶ MilitaryZone
▶ Muzzle
▶ PirateCoin
GUI Image Name
0
Layer To add
Add GUI Image
▶ Add GUI Button
▶ Add GUI Animation
▶ Custom Animations


Editor
Switch to GUI selection mode
Add New Object
Model Groups
Add Trigger Volume
Add New Light
Add GUI Elements
Add GUI Layer
Add GUI Text
Add GUI Image
Add GUI Button
GUI Button Name
Filter Assets
crosshair.png
crosshair_.png
damageIndicator.png
Dwarf
EasyFPS
Health-icon.png
LoadImages
Mario
MayanTemple
Menu
Buttons
cancelC.png
cancelD.png
cancelH.png
cancelN.png
disabled.png
newGameC.png
newGameD.png
Normal image Set
On hover image Set
On click image Set
Disabled image Set
0
Layer To add
Add GUI Button (?)



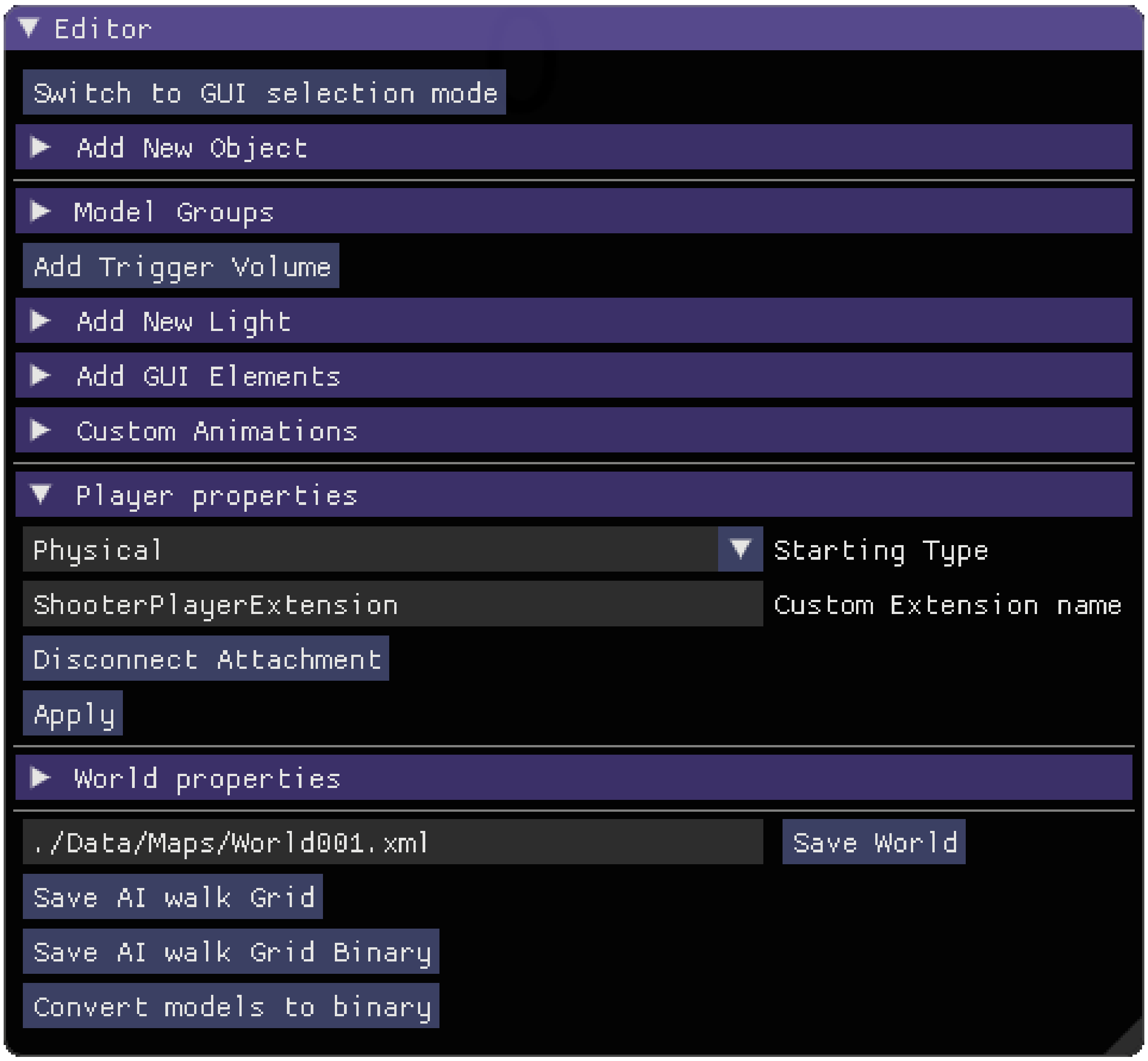
GUI Animation Name

60 FrameSpeed

Is Animation Looped

Filter Assets

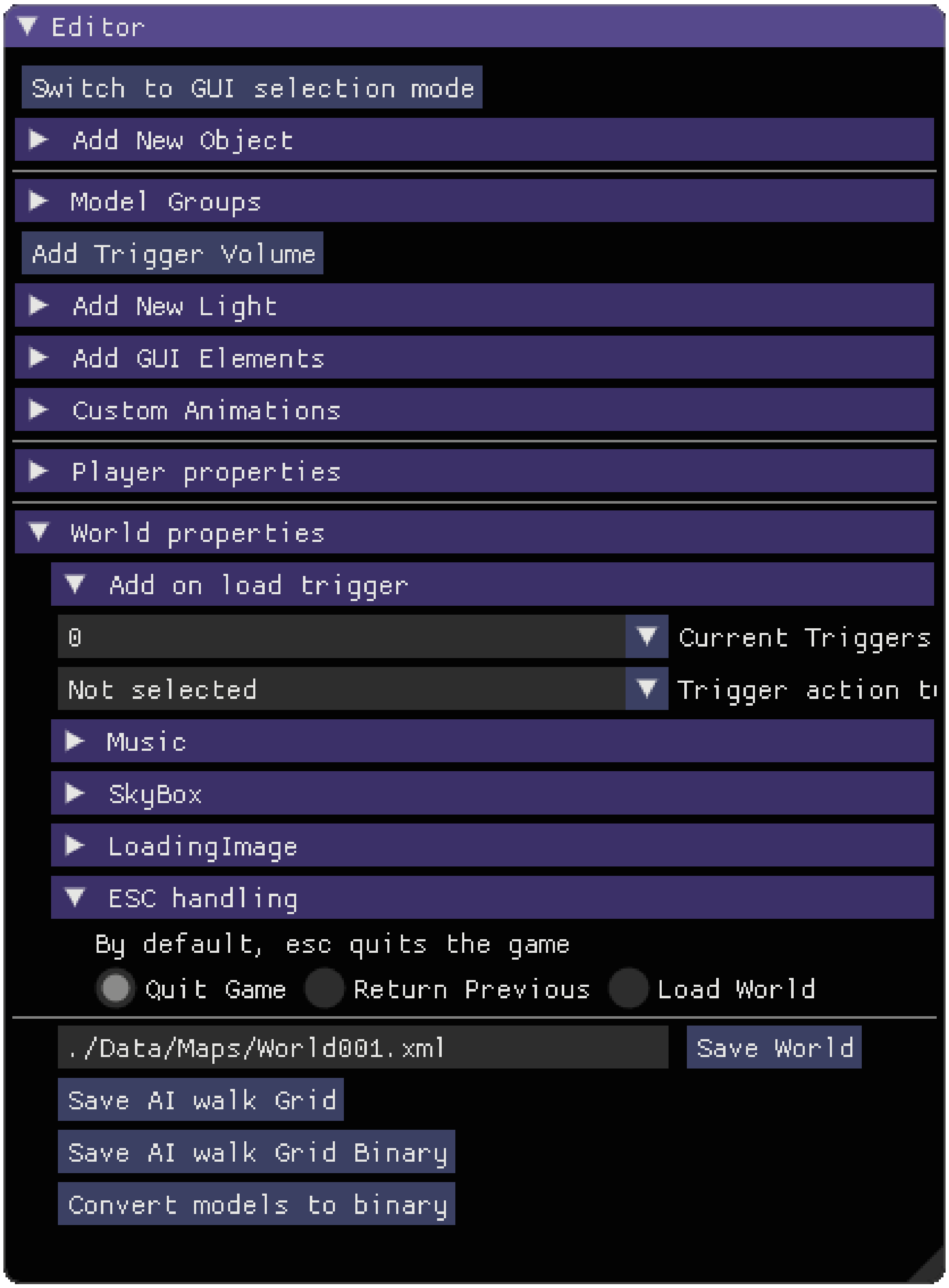
0 Layer To add



Physical Starting Type

ShooterPlayerExtension Custom Extension name

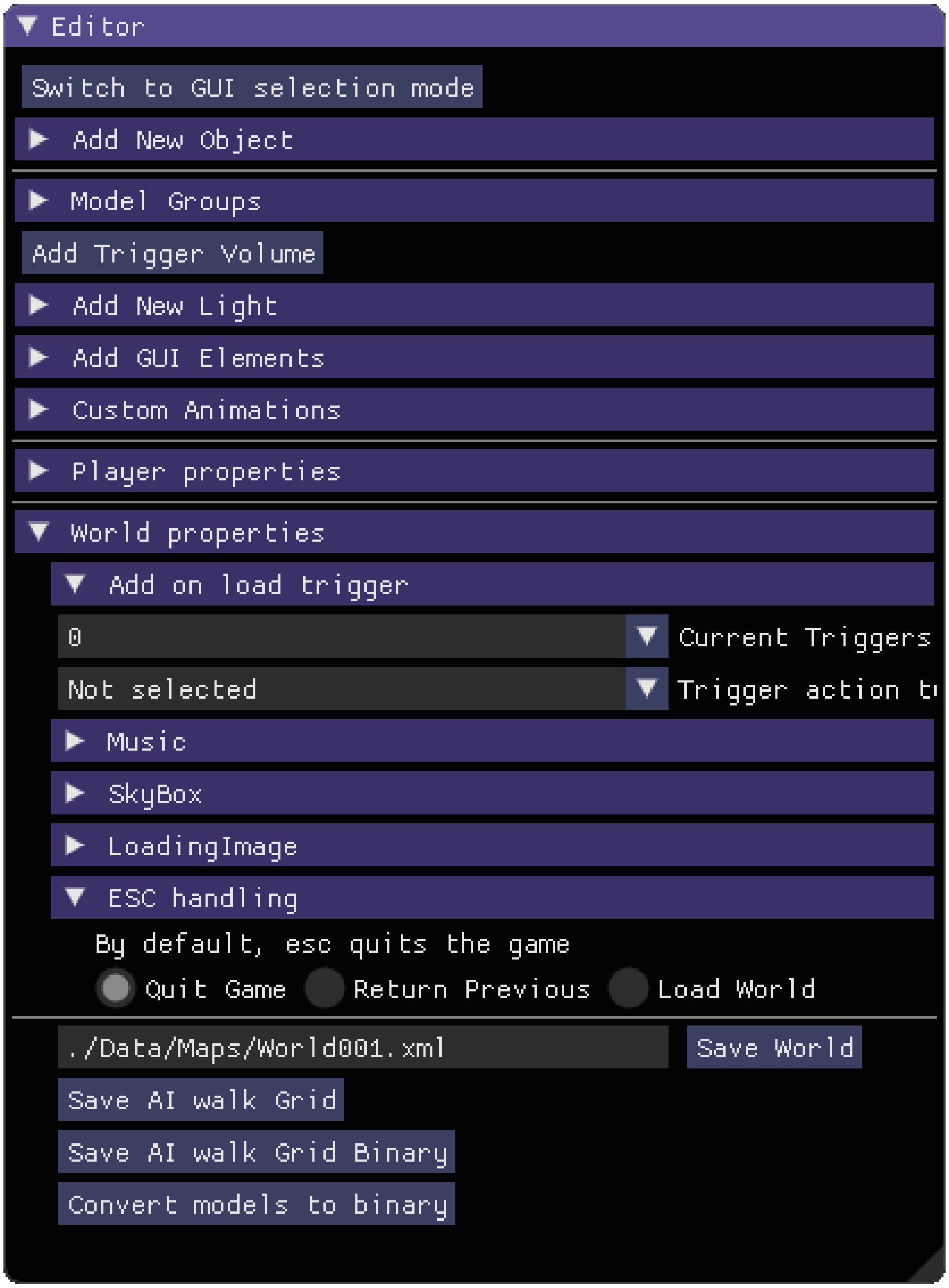
./Data/Maps/World001.xml



By default, esc quits the game

[✓ Quit Game] Return Previous Load World

./Data/Maps/World001.xml



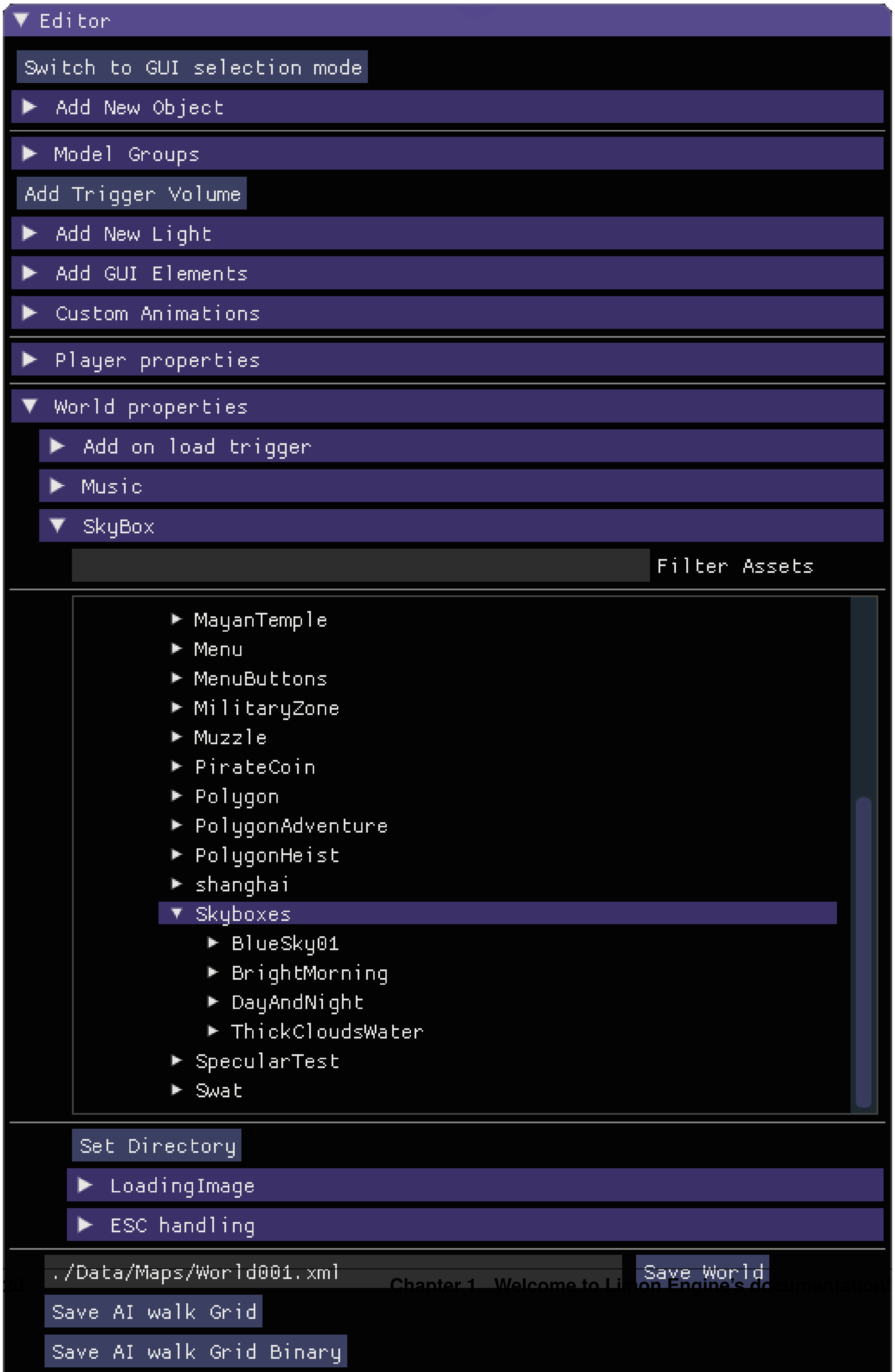
0

Not selected

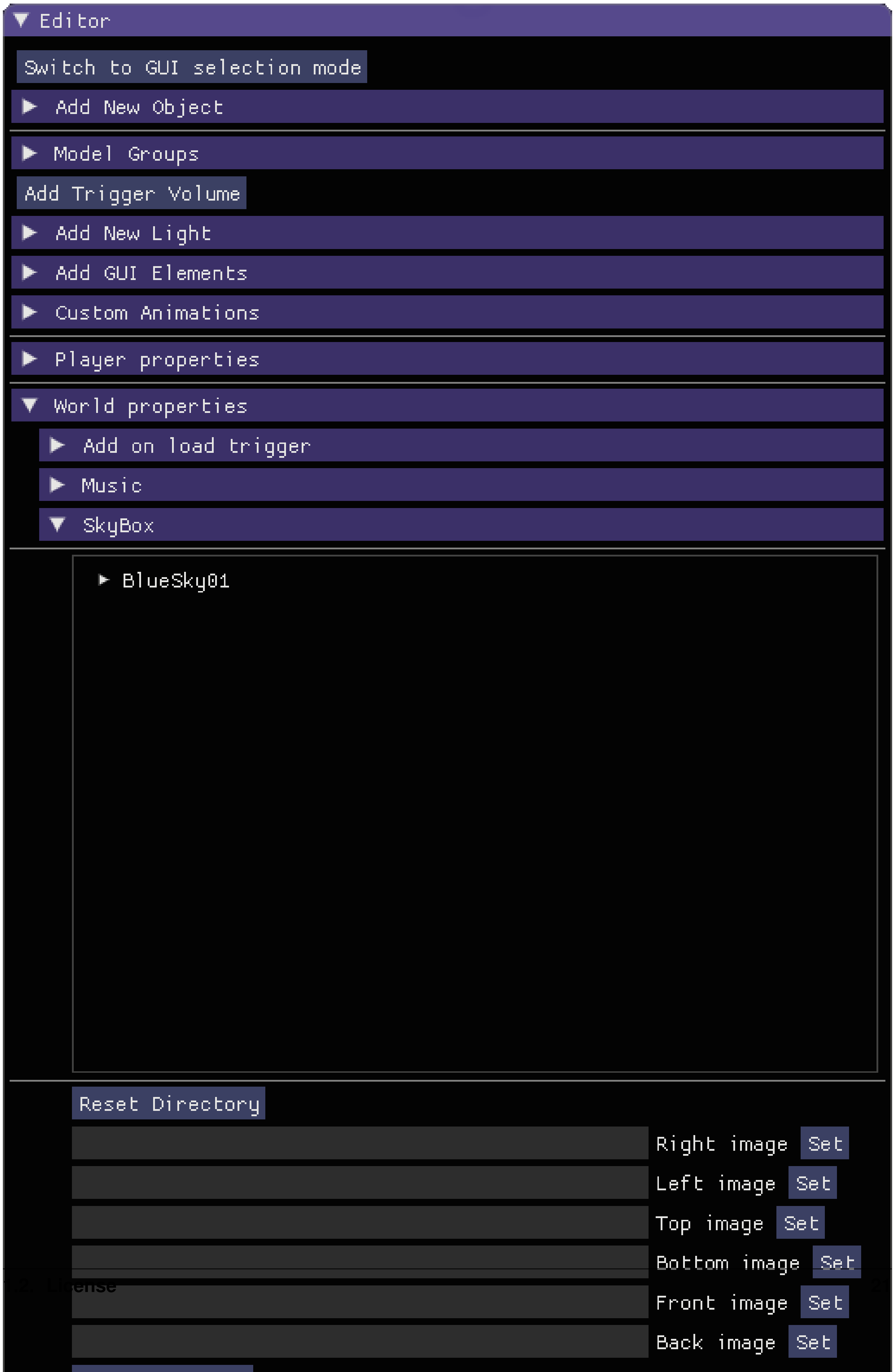
By default, esc quits the game

[✓ Quit Game] Return Previous Load World

./Data/Maps/World001.xml



./Data/Maps/World001.xml


Editor
Switch to GUI selection mode
Add New Object
Model Groups
Add Trigger Volume
Add New Light
Add GUI Elements
Custom Animations
Player properties
World properties
Add on load trigger
Music
SkyBox
BlueSky01
Reset Directory
Right image Set
Left image Set
Top image Set
Bottom image Set
Front image Set
Back image Set

Setting skybox is two step process. First directory that contains the images is set, then 6 image that form up the skybox will be selected.

Loading Image

A loading image can be set using the loading image directory listing. If no image is set, an empty screen will be shown.

ESC Handling

This setting allows customising the behaviour of ESC key.

- Quit Game: exits the game immediately without asking for a verification
- Return Previous: Loaded maps list is kept within the engine. This option returns the world before current one. If this is the first world, or this world is loaded with force new directive, this option does nothing.
- Load World: This option add a new text input to the editor. The map at the path entered will be loaded if not already, and the current map will switch to the entered one.

Other editor controls

Loaded custom animations will be listed under custom animations for convenience. You can load other custom animation by entering the file path.

Saving the map

The map will be saved at the path when save world is clicked, overriding if it already exists.

Limon Auto generates a walking grid for path finding, used by AI Actors. Generating such data takes minutes on big maps, so saving it with map is a must for load speed of a map. There are 2 types of saving supported for this grid, binary and XML. XML should be avoided by game developers, it is only useful for engine developers. XML format can take up to 2GB of memory. Binary format is the go-to format.

Convert models to binary button scans all the models used in the map, and converts used assets to "*.limonmdel" files. This files use less ram, faster to load and not reversable, making them suitable for game releases. Map itself will be updated to use them if this button is pressed.

Warning: It is worth repeating. The save button overrides if there is a file with same name. Please pay attention.

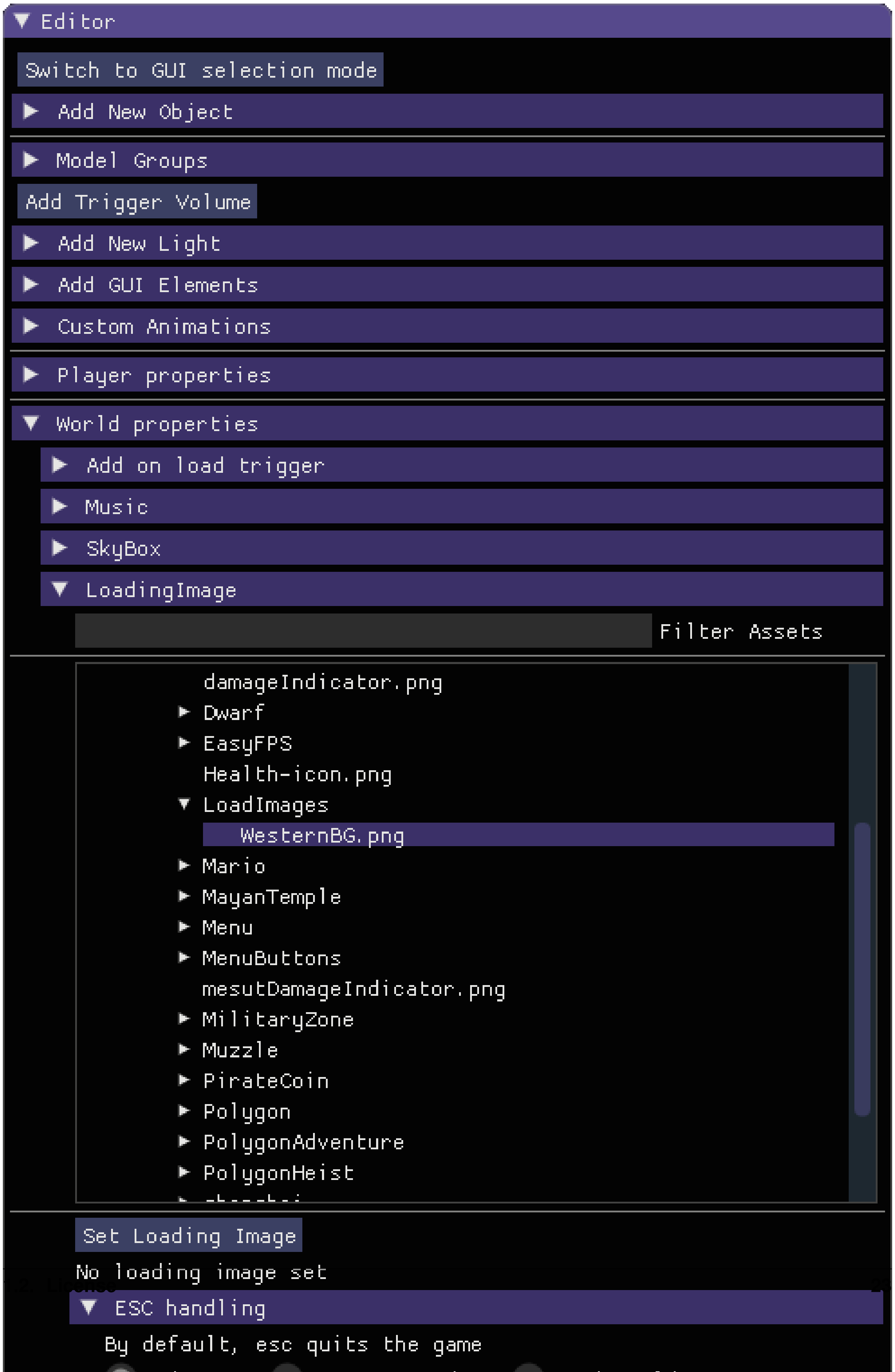
Object Editor Details

Object editor has two parts. One is the window that is on the left by default, and the other is the gizmos that appear at the position of the object. The window content changes based on the selected object. Each possible object type is documented separately below.

Object editor has tree view of the map, you can use it to change selected object, as well as clicking on them. All object have a remove button at the end of the window, which removes selected object completely.

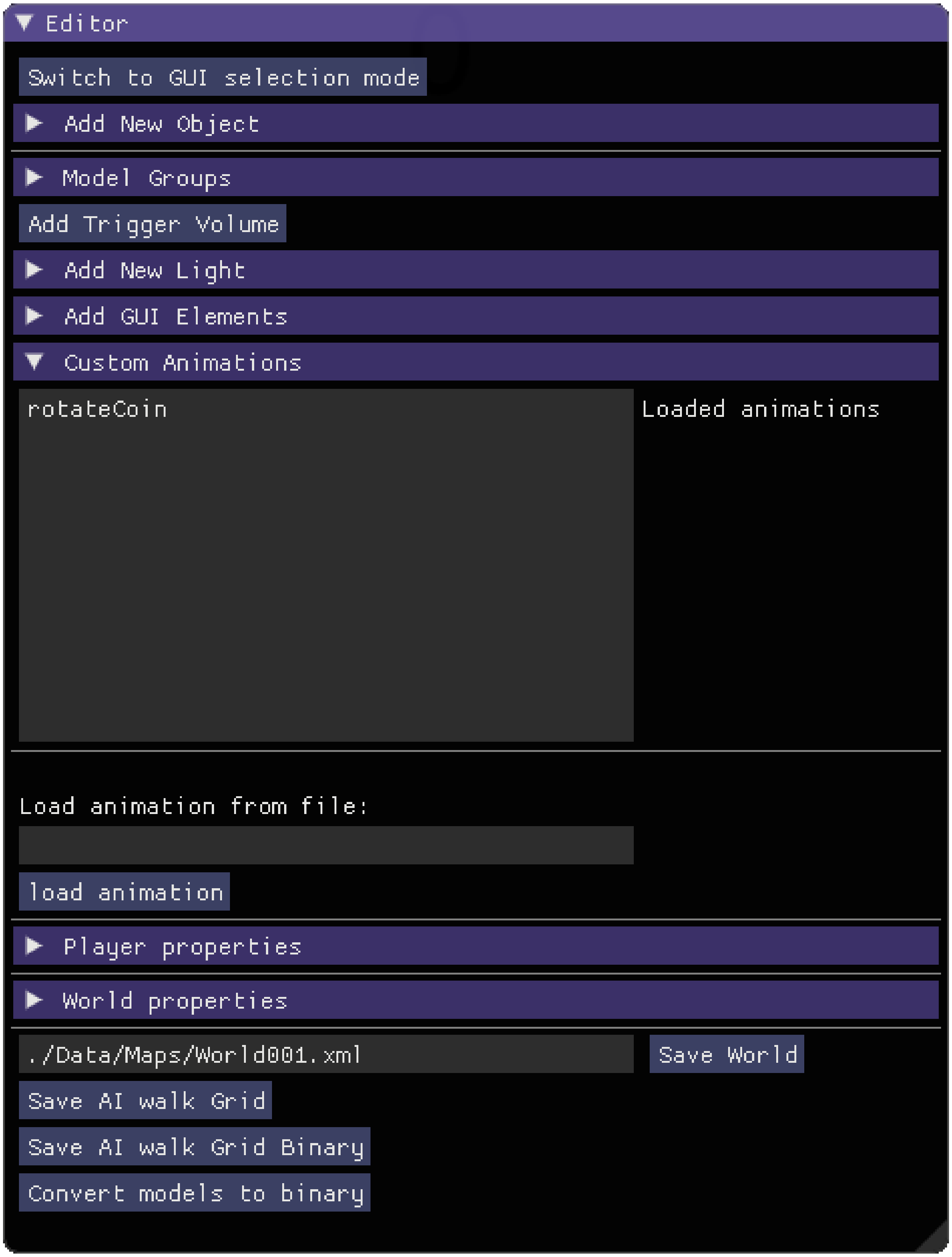
Model Object Settings

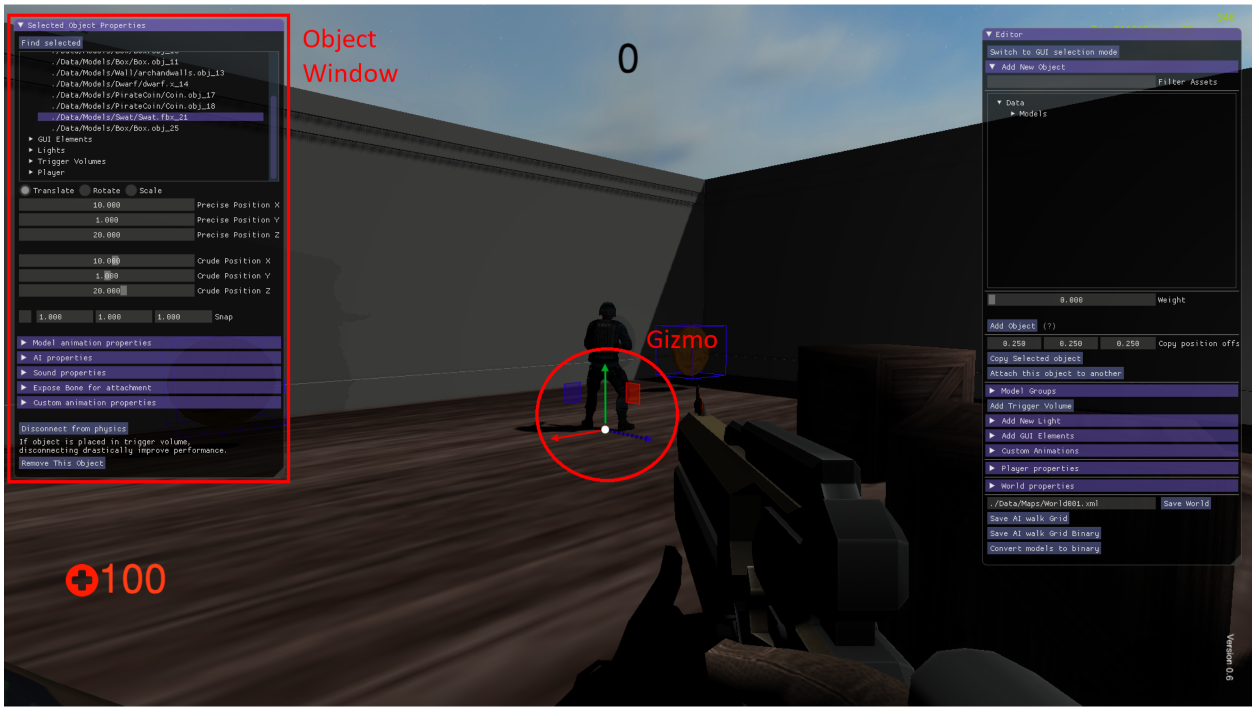
There are 3 radio buttons under the selected object Drop-down. These are "Translate", "Rotate", "Scale". Based on the selected mode, the 6 elements below change, but their usage is the same. First 3 are used for precise settings by



No loading image set

By default, esc quits the game


Editor
Switch to GUI selection mode
Add New Object
Model Groups
Add Trigger Volume
Add New Light
Add GUI Elements
Custom Animations
rotateCoin
Loaded animations
Load animation from file:
load animation
Player properties
World properties
./Data/Maps/World001.xml
Save World
Save AI walk Grid
Save AI walk Grid Binary
Convert models to binary



10.000 Precise Position X
1.000 Precise Position Y
20.000 Precise Position Z

10.000 Crude Position X
1.000 Crude Position Y
20.000 Crude Position Z

1.000 1.000 1.000 Snap

If object is placed in trigger volume,
disconnecting drastically improve performance.

0.000 Weight

0.250 0.250 0.250 Copy position offs

./Data/Maps/World001.xml

dragging, or entering exact value by typing. **To enter typing mode, you should double click the item.** The second 3 items are for setting the values with bigger differences.

Just under these settings, there is snap settings. It is used by gizmo. For details check *Gizmo Usage*.

If the loaded model contains animations within, these animations are listed under the "Model animation properties", and the speed of this animation can be set using "Animation time scale". If a new animation as part of old one is needed, "Seperate selected animation by time" part can be used. It takes 2 floats as input, and a name. After Create section button is clicked, the animation will be listed just as others.

Limon also supports loading animations from another file, specifically to support Mixamo.com animations. When a model is loaded, Limon checks if there is a "Mixamo" directory at the same path, and if it exists, try to loads the files in that directory as animations for model.

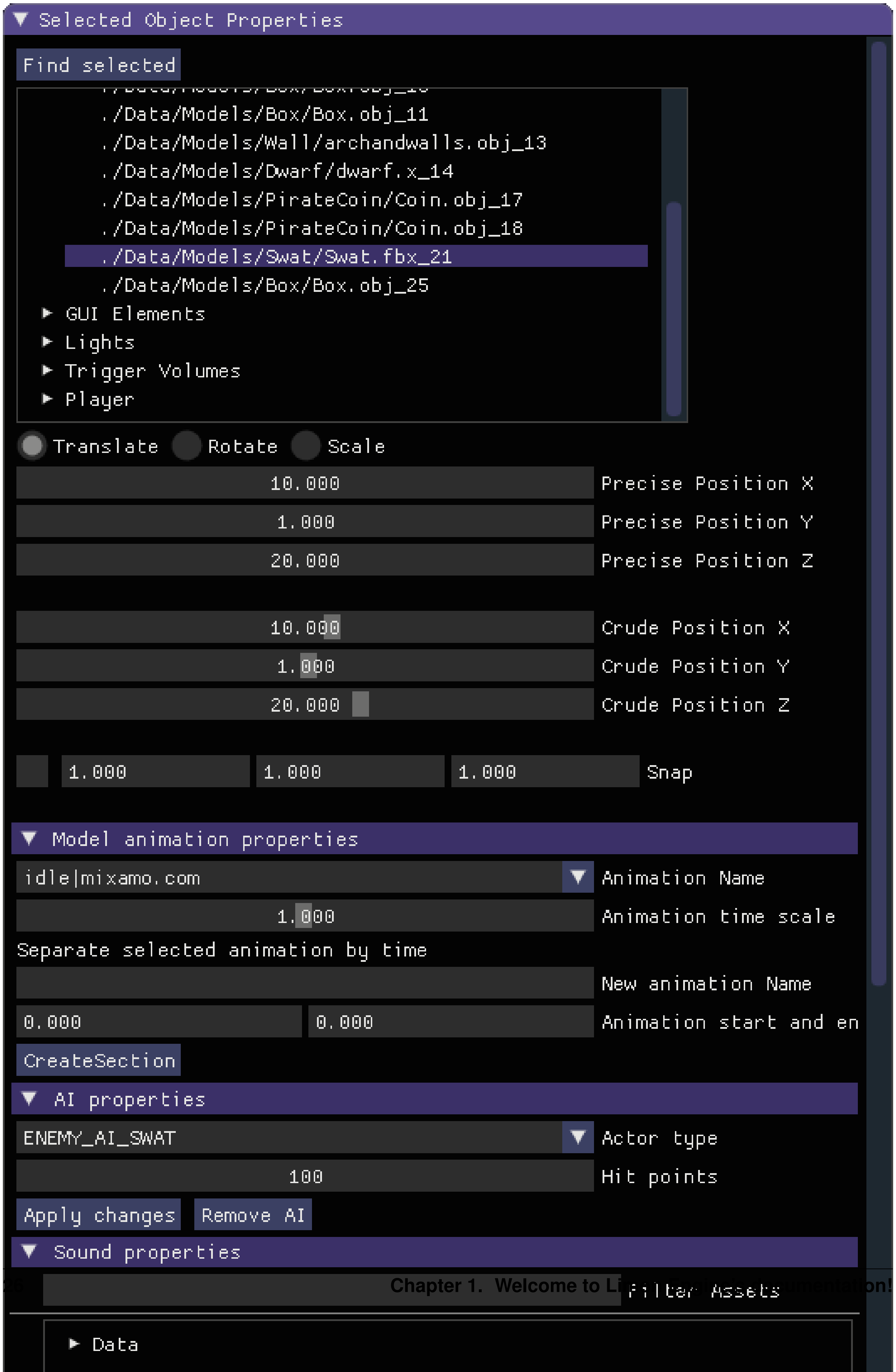
AI properties section has a list of available Actors. Game developers can write custom actors using API provided. Please check *How to Implement an AI Actor* for details. If selected actor has settings exposed, they will be listed under actor type drop-down.

Note: If model has no animation, it can't be assigned an AI. Both Animation properties and AI properties will be hidden in that case.

Under AI settings, there is "Step on Sound" setting. This is used as step sound when "Physical player" move on top of the model.

After That there is "Custom animation properties". This section lists currently available animations, you can apply any of the custom animations to any number of models. If you want to create a new custom animation, you can do so by using "Create new" button. This button will open animation sequencer. For details please check TriggerVolumes.

Disconnect from physics button removes the collision mesh from map so the object won't be interacting with physics engine. This can be useful for small probes that should be ignored.



▼ Selected Object Properties

Find selected

./Data/Models/Box/Box.obj_11
./Data/Models/Wall/archandwalls.obj_13
./Data/Models/Dwarf/dwarf.x_14
./Data/Models/PirateCoin/Coin.obj_17
./Data/Models/PirateCoin/Coin.obj_18
./Data/Models/Swat/Swat.fbx_21
./Data/Models/Box/Box.obj_25
▸ GUI Elements
▸ Lights
▸ Trigger Volumes
▸ Player

● Translate ○ Rotate ○ Scale

10.000	Precise Position X
1.000	Precise Position Y
20.000	Precise Position Z
10.000	Crude Position X
1.000	Crude Position Y
20.000	Crude Position Z

1.000 1.000 1.000 Snap

▼ Model animation properties

idle|mixamo.com	Animation Name
1.000	Animation time scale

Separate selected animation by time

	New animation Name
0.000 0.000	Animation start and en

CreateSection

▼ AI properties

ENEMY_AI_SWAT	Actor type
100	Hit points

Apply changes Remove AI

▼ Sound properties

Filter Assets

▸ Data

Trigger Object Settings

The trigger object has same interfaces with model for transformation settings. The difference is at "Trigger Properties" section.

This section has 3 Trigger settings.

1. First Enter Trigger.
2. Enter Trigger.
3. Exit Trigger.

The details of Triggers settings are not predefined, triggers can define their own settings. For details, please refer to Triggers. Any or all of the triggers can be left unset.

The logic of triggers is as follows:

1. If player is not detected, and wasn't detected last frame, do nothing.
2. If player is not detected, and was detected last frame, and *Exit Trigger* is set, run it.
3. If player is detected, and was detected last frame, do nothing.
4. If player is detected, and wasn't detected last frame:
    1. If player was not detected ever before, and *First Enter Trigger* is set, run it.
    2. If player was not detected ever before, but *First Enter Trigger* is not set, and *Enter Trigger* is set, run *Enter Trigger*.
    3. If player was detected before, if *Enter Trigger* is set, run *Enter Trigger*.

GUI Text Settings

GUI Text has custom name that can be updated using the name field. This field doesn't allow spaces of any kind.

The text to render is set using *Text* field.

Position X and Position Y is used for transformation of the text, and Color R G B are the text color.

GUI Image Settings

GUI Image has custom name that can be updated using the name field. This field doesn't allow spaces of any kind.

The File is the path to image file. Changes on this field is only applied when change image button is clicked.

Full screen click box scales the image to fill the screen, and disables scale and position settings.

Position and Scale are used to set the Transform of the image.

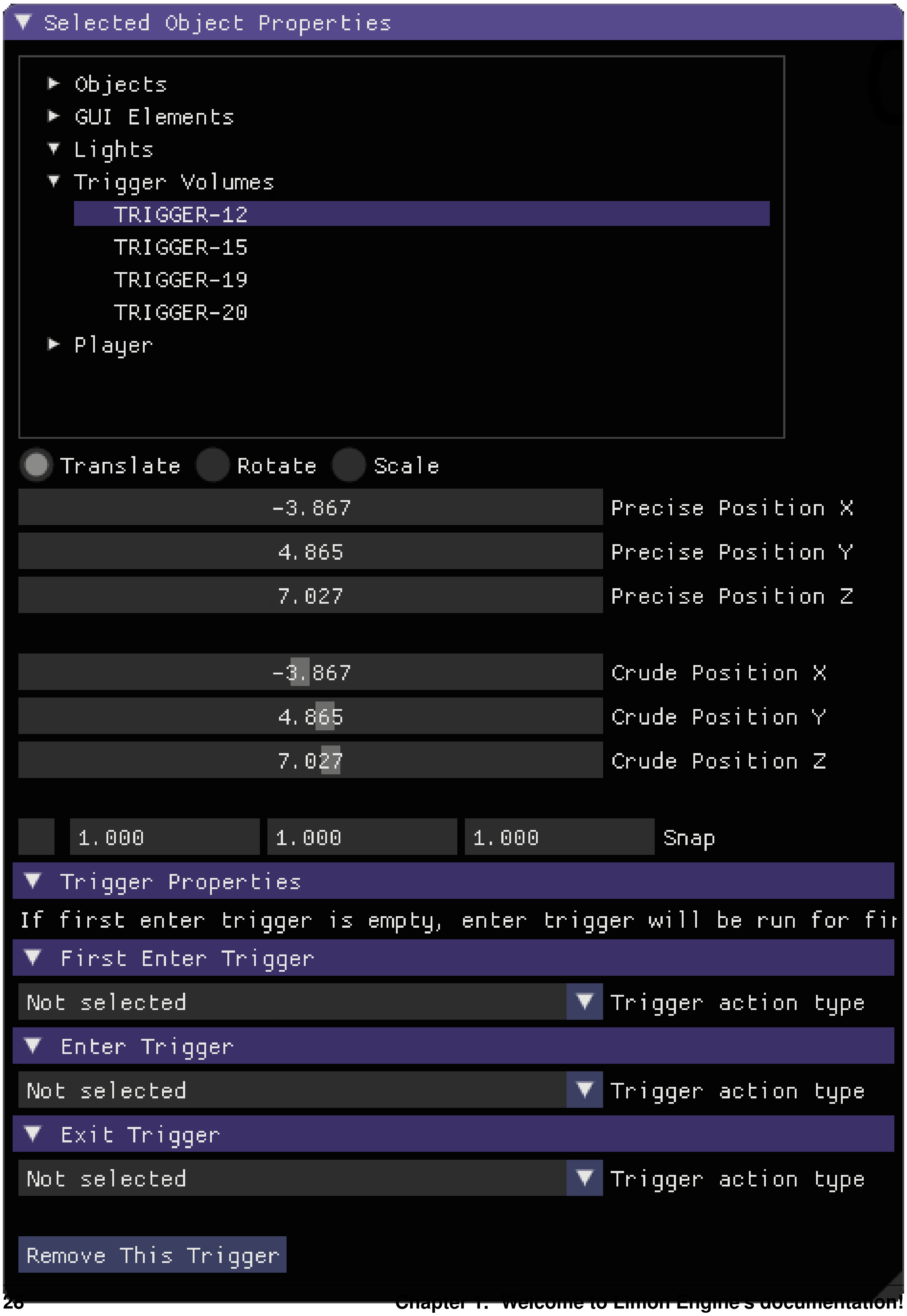
GUI Button Settings

GUI Button has custom name that can be updated using the name field. This field doesn't allow spaces of any kind.

There are 4 file settings. Only the Normal file is required, the rest are optional.

The Button can be interactive, depending on the player state. If player is set interactive, the following logic is used:

1. If The button doesn't have an trigger, and have a disabled image set, the disabled image will be shown.



▼ Selected Object Properties

▶ Objects
▶ GUI Elements
▼ Lights
▼ Trigger Volumes
TRIGGER-12
TRIGGER-15
TRIGGER-19
TRIGGER-20
▶ Player

● Translate ○ Rotate ○ Scale

-3.867 Precise Position X
4.865 Precise Position Y
7.027 Precise Position Z

-3.867 Crude Position X
4.865 Crude Position Y
7.027 Crude Position Z

1.000 1.000 1.000 Snap

▼ Trigger Properties

If first enter trigger is empty, enter trigger will be run for fir

▼ First Enter Trigger

Not selected Trigger action type

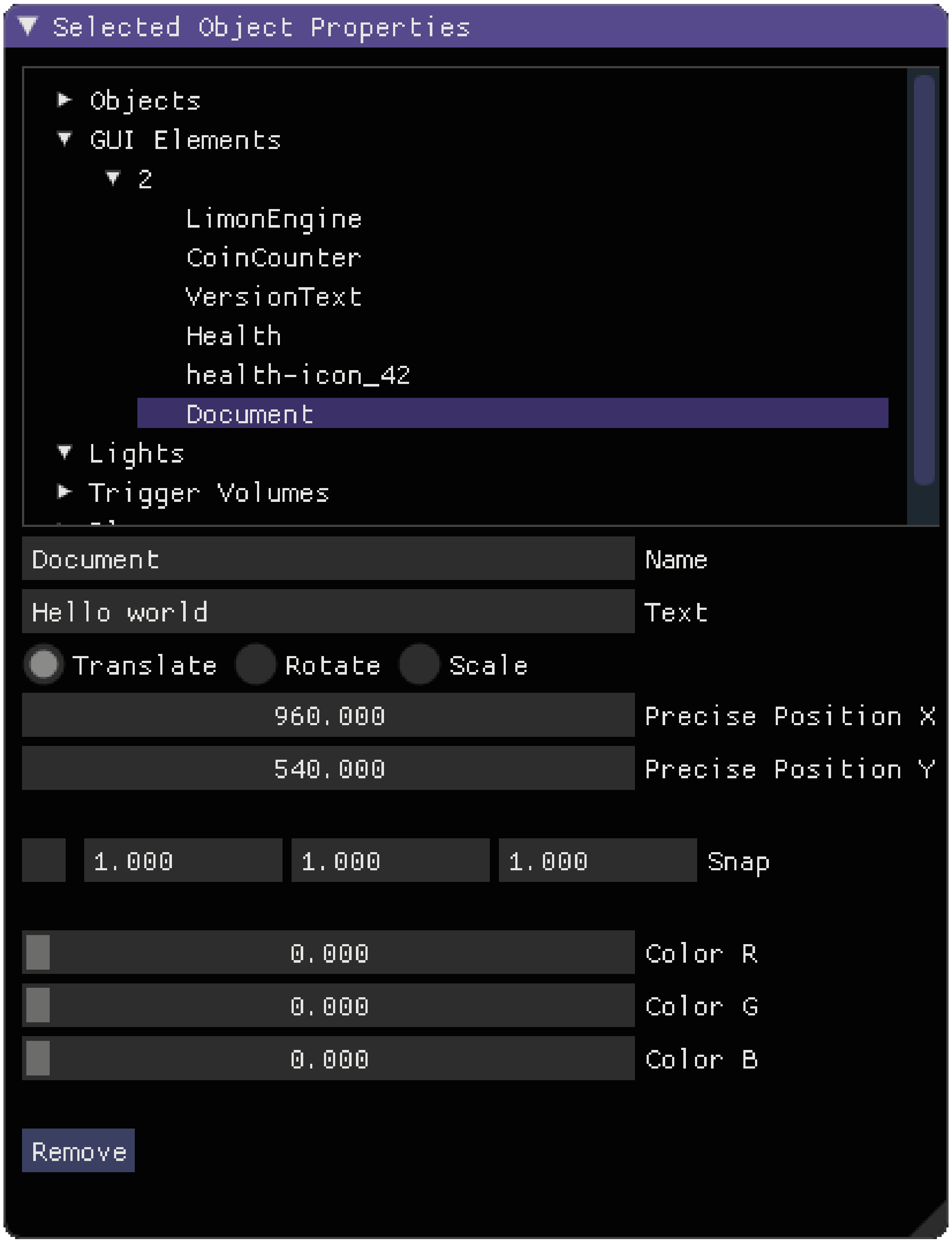
▼ Enter Trigger

Not selected Trigger action type

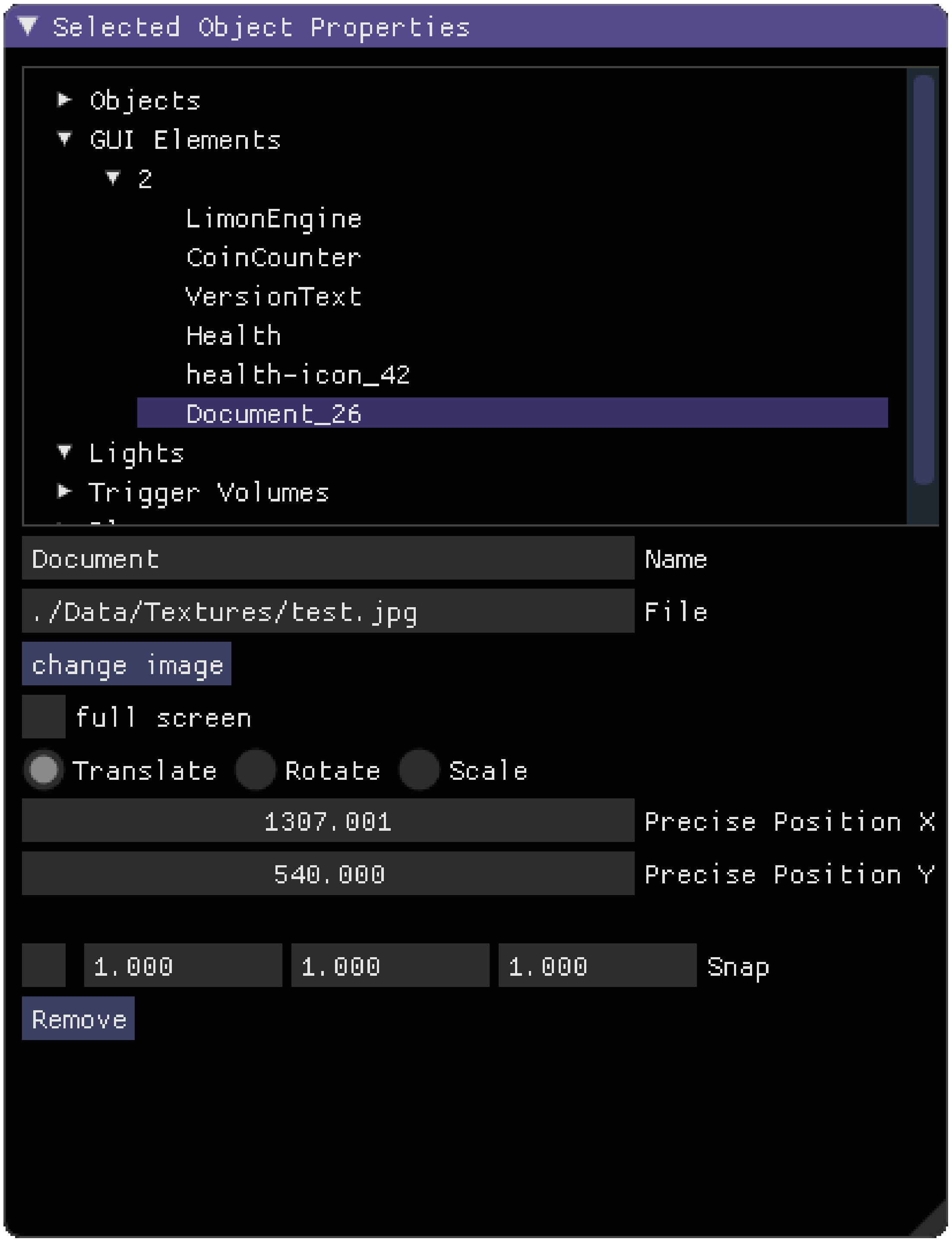
▼ Exit Trigger

Not selected Trigger action type

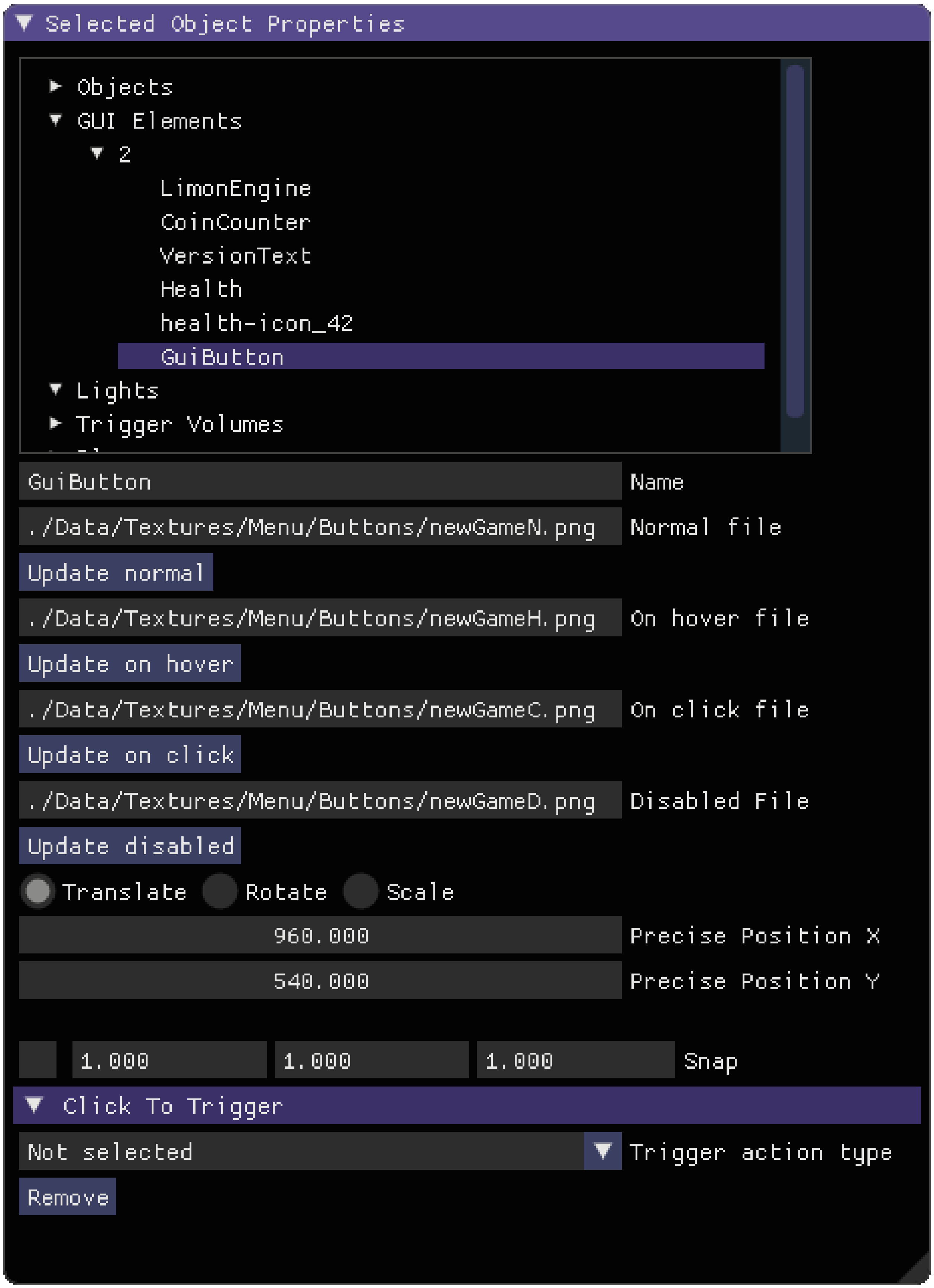
Remove This Trigger



```
Document                     Name
Hello world                  Text
● Translate  ○ Rotate  ○ Scale
960.000                      Precise Position X
540.000                      Precise Position Y

[ ] 1.000  1.000  1.000      Snap

0.000                        Color R
0.000                        Color G
0.000                        Color B

Remove
```



Name: Document
File: ./Data/Textures/test.jpg
full screen
Precise Position X: 1307.001
Precise Position Y: 540.000
Snap: 1.000 1.000 1.000



Selected Object Properties

- Objects
- GUI Elements
  - 2
    - LimonEngine
    - CoinCounter
    - VersionText
    - Health
    - health-icon_42
    - GuiButton
- Lights
- Trigger Volumes

GuiButton	Name
./Data/Textures/Menu/Buttons/newGameN.png	Normal file
Update normal	
./Data/Textures/Menu/Buttons/newGameH.png	On hover file
Update on hover	
./Data/Textures/Menu/Buttons/newGameC.png	On click file
Update on click	
./Data/Textures/Menu/Buttons/newGameD.png	Disabled File
Update disabled	

[✓ Translate] Rotate Scale

960.000	Precise Position X
540.000	Precise Position Y

1.000 1.000 1.000 Snap

Click To Trigger

Not selected	Trigger action type

Remove

2. If on click image is set, and mouse is down over the button, on click image is shown. Also Trigger will be run. For details, please check Triggers.
3. If on hover image is set, and mouse cursor is over the button, that image will be shown.
4. If all else were wrong, the normal image will be shown.

Position and Scale are used to set the Transform of the button.

Trigger section allows to set the trigger to run when clicked.

Just under these settings, there is snap settings. It is used by gizmo. For details check *Gizmo Usage*.

Gizmo Usage

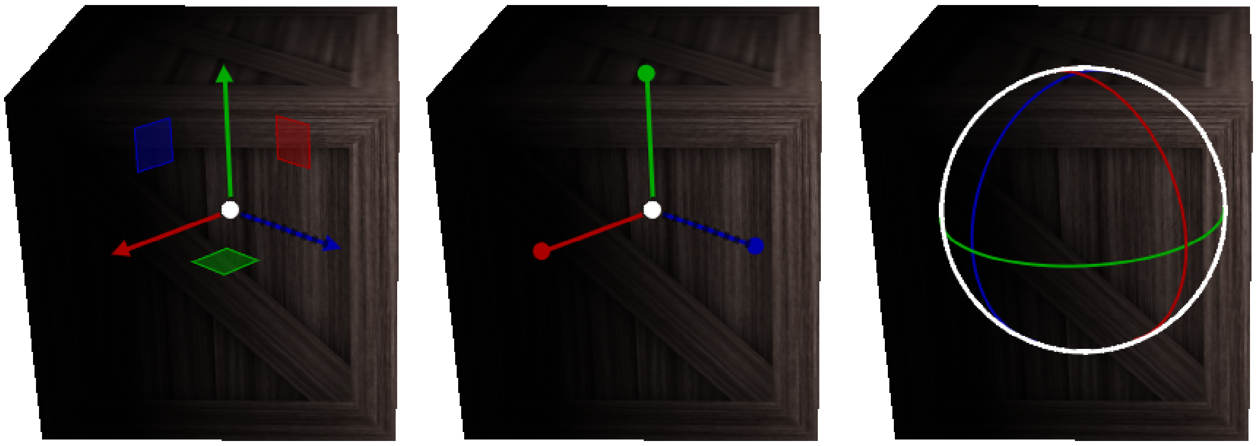


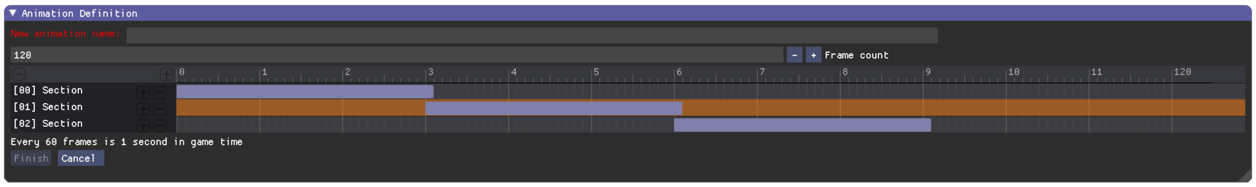
Fig. 5: Gizmo types: Translate, Scale, Rotate

The gizmo is the tool interface that appears at the position of the object that is selected. It has 3 modes, translate(move), scale and rotate. These modes are set using the object editor window, and not all of them are available for all object types. They are directly attached to the editor information, so change in one will update the other.

All three modes use same logic. Dragging an axis applies the transform on that axis. Meaning while in translate mode, clicking on vertical line and dragging will move model vertically. Dragging by the center moves freely, without axis locking. Translate mode also has boxes that can be used to move on a plane, instead of a line.

Some objects have an *snap* setting. This setting is used by the gizmo, to determine step size of the update. Snap of 0.25 in scale mode means dragging the gizmo will scale the object as 1, 1.25, 1.5, 1.75 etc. Same applies for translate and rotate too.

Animation Sequencer Details



Limon Engine can be used to animate objects or GUI elements using all three transformations. You can select the object you want to animate, and use the custom animation section. If you choose to create a new custom animation, the animation sequencer will be shown.

The sequencer starts with animation name. You can't save/finish an animation without a name. If animated object is a GUI element, translate z component will be used by alpha channel, enabling animating alpha changes.

Warning: If an animation with the same name exists, the old one will be overriden.

Warning: Animations for GUI elements and 3D objects can be used interchangeably, but the results might not be as predicted.

Second line is how many frames long the animation will be. After that the main sequencer part comes. The "-" symbol is used to hide/show the sequences. "+" sign is used to add a section to the end of the animation. Each section has "+" and "-" symbols next to them. "-" removes the section, and "+" add a section just after the selected one, with the same length.

Note: Animations are considered 60 frames per second.

How to create animations

When sequencer is shown, it will have 1 section. You can imagine the sections, as "What is the final transform, after given time". You can move the object around, scale, rotate as you wish, to set the transform. THe length of the section is time. If you want the animation to have multiple sections, you can press the either "+" button to add another section. When you hit finish, the animation will be saved and applied.

Note: The custom animation is assumed to be built for the selected object, and to run in a loop, starting with map load. If this is wrong, you can remove the animation from the object.

Extending Engine Capabilities

Limon API Usage

Limon Engine provides an C++ API for extending and customising it to fit your game. The API has a parameter system for requesting and providing variables, and the parameters are connected to both editor and serialize/deserialize subsystem so saving and loading is handled by the engine.

LimonAPI class

The LimonAPI class has all the methods available for usage. It also provides means to pass data around, namely ParameterRequest struct. This struct is used for both asking for and providing data. This struct is de/serialized by the engine, and editor can build graphical interfaces for it, there is no need to worry about that aspects of development.

ParameterRequest struct

The struct is main means of data transfer. The serialize and deserialize methods are meant to be used by engine internals, they should be ignored for API usage purposes.

Note: The value of any instance is initialized to 0.

RequestParameterTypes Enum

Used to indicate the semantic meaning of the parameter. Editor will render interface accordingly. The *requestType* variable keeps the value.

Possible values:

- MODEL: Lists the models in the map. Uses variable type LONG, sets handle id.
- ANIMATION: Lists the custon animations loaded in the map. Uses variable type LONG, sets handle id.
- SWITCH: Renders tick box. Uses variable type BOOLEAN, sets is ticked.
- FREE_TEXT: Renders input box. Uses variable type STRING. Sets the input text.
- TRIGGER: Lists the trigger volumes in the map, and selector for "first enter", "enter", "exit". Uses variable type LONG_ARRAY, sets handle id, and selected trigger. for details refer to *Trigger Object Settings*
- GUI_TEXT: Lists the GUI Text elements in the map. Uses variable type LONG, sets handle id
- FREE_NUMBER: Renders input box for number. Uses variable type LONG. Sets the number entered.
- COORDINATE: Used to pass 3D vectors. Editor doesn't handle this type.
- TRANSFORM: Used to pass 4x4 Transformation matrix. Editor doesn't handle this type.
- MULTI_SELECT: Renders combobox. In a request, same description and multi select parameters are grouped to build the combobox. First of this group is the selected object. Selected object should be repeated at its desired position. Ex: "apple, banana, apple, grape" values will render "banana, apple, grape" combobox with apple selected.

ValueTypes Enum

Used to determine how to handle Value union. *valueType* variable keeps the value.

Possible values:

- STRING
- DOUBLE
- LONG
- LONG_ARRAY
- BOOLEAN
- VEC4
- MAT4

Description String

Used in editor and shown to user directly. It is also used to group MULTI_SELECT elements to build the combobox.

Value Union

Union to store value set by editor or other triggers or engine itself. *value* variable keeps it.

Union variables

- char stringValue[64]
- long longValue
- long longValues[16]
- double doubleValue
- bool boolValue
- Vec4 vectorValue
- Mat4 matrixValue

Note: if long values array is used, first element should be used element count.

isSet

Used to indicate if the variable is set or or not. If default value is considered valid then should be initialized true.

Warning: If a variable is not required aka optional, this should be initialized with true, because editor doesn't allow saving a trigger with any parameter not set.

How to Implement an Action

Actions are generalized by the class TriggerInterface, under src/GamePlay of the engine. Each new action must implement this interface. These actions can be assigned to trigger volumes or GUI button.They can also be run on map load.

Action constructors will get a populated LimonAPI instance for the world they are to be run in. Using this instance, API calls can be made to interact or change the world to game design.

Actions never get persisted, but parameters to pass to actions are persisted with maps.

TriggerInterface Class

	*TriggerInterface(LimonAPI *limonAPI)*
std::vector<LimonAPI::ParameterRequest>	*getParameters()*
bool	*run(std::vector<LimonAPI::ParameterRequest>parameters)*
std::vector<LimonAPI::ParameterRequest>	*getResults()*
std::string	*getName() const*

TriggerInterface(LimonAPI *limonAPI)

The constructor of the interface.

Note: All actions must have the same signature, no other parameters should be required.

getParameters()

Returns a vector of *ParameterRequest struct*, These parameters are going to be set by map designer using the editor.

run(std::vector<LimonAPI::ParameterRequest>parameters)

The parameters with their set values will be provided. The logic of the action should be this method. Return true if run succesfully. Return false if the run failed for some reason.

getResults()

The actions result might be queried by other actions. This method should return the results. Engine itself doesn't use this method, so it can return an empty vector. The usage of this method is game specific.

For example if the action adds a GUI element, and another action wants to remove this element, the other action might query for gui element id.

getName() const

Returns the name of the action.

Warning: The name must be unique, or the results will be undefined.

How to enable Dynamic Library discovery

Limon engine will try to load custom actions on engine startup, from libcustomTriggers file (extension based on platform). If the file is found, engine will check for a method with following signature:

```
void registerAsTrigger(std::map<std::string, TriggerInterface*(*)(LimonAPI*)>*␣
↪triggerMap)
```

This method should fill the triggerMap passed, with all the custom actions, like this:

```
(*triggerMap)["$ACTION_NAME1$"] = &createT<$ActionClass1$>;
(*triggerMap)["$ACTION_NAME2$"] = &createT<$ActionClass2$>;
```

How to Implement a Player Extension

Player extensions are main ways of handling input in Limon Engine. Input from player is first handled by PhysicalPlayer class, which governs look around and movement, then all input information is passed to selected extension, so it can handle custom interactions, like pickups, shooting etc. On the other side, any interaction send by other entities to player is directly passed to player extension for handling.

PlayerExtensionInterface Class

PlayerExtensionInterface(LimonAPI *limonAPI)

The constructor of the interface.

Note: All Player Extension must have the same signature, no other parameters should be required.

void processInput(const InputStates &inputState, long time)

Called each frame with updated input information, and time of frame in milliseconds.

void interact(std::vector<LimonAPI::ParameterRequest> &interactionData)

Called by other entities to interact with player.

std::string getName() const

Returns the name of the Player Extension.

Warning: The name must be unique, or the results will be undefined.

InputStates Class Usage

InputStates is a thin wrapper around SDL2 input events. It has 4 main methods that can be used:

1. getInputStatus: Allows checking if a key is down or up, for keys used by engine. 3 buttons of mouse is included.
2. getInputEvents: Allows if a key state changed in last frame, for keys used by engine. 3 buttons of mouse is included.
3. getRawKeyStates: Allows to check all key states for current frame.
4. getMouseChange: Allows checking for relative or absolute position of the mouse, depending of the mode(Menu player uses absolute).

Full list of supported keys can be checked from src/InputStates.h

How to enable Dynamic Library discovery

Limon engine will try to load custom player extensions on engine startup, from libcustomTriggers file (extension based on platform). If the file is found, engine will check for a method with following signature:

```
void registerPlayerExtensions(std::map<std::string,␣
↪PlayerExtensionInterface*(*)(LimonAPI*)>* playerExtensionMap)
```

This method should fill the actorMap passed, with all the custom actors, like this:

```
(*playerExtensionMap)["$EXTENSION_NAME1$"] = &createPlayerExtension<$ExtensionClass1$>
↪;
(*playerExtensionMap)["$EXTENSION_NAME2$"] = &createPlayerExtension<$ExtensionClass2$>
↪;
```

How to Implement an AI Actor

AI Actor is the way Limon enables custom behaviour for NPCs or enemies. Limon engine has a class named ActorInterface under src/AI, that it used to implement custom AI behaviour.

Limon editor allows selecting Actor per model. After Actor selected, a new instance of the selected Actor is created, and called each frame with current information about the world. It is also possible to expose some settings to be filled by level designer using the same interface of editor.

ActorInterface Class

ActorInterface class has two helper structs used to pass information between engine and AI. Those are *ActorInformation struct* and *InformationRequest struct*. details are below.

Warning: Information requests are prepared by separate threads, and no guarantees made for when they will return. Check routeReady flag on ActorInformation before using the route.

	*ActorInterface(uint32_t id, LimonAPI *limonAPI)*
std::vector<LimonAPI::ParameterRequest>	*getParameters()*
void	*setParameters(std::vector<LimonAPI::ParameterRequest>parameters)*
void	*play(long time, ActorInformation& information)*
bool	*interaction(std::vector<LimonAPI::ParameterRequest>parameters)*
std::string	*getName() const*

ActorInterface(uint32_t id, LimonAPI *limonAPI)

The constructor of the interface.

Note: All Actors must have the same signature, no other parameters should be required.

std::vector<LimonAPI::ParameterRequest> getParameters()

Returns a vector of *ParameterRequest struct*, These parameters are going to be set by map designer using the editor. These parameters should be filled with their current values, because Load/Save logic uses these parameters to persist AI information.

void setParameters(std::vector<LimonAPI::ParameterRequest> parameters)

The parameters set by map designer will be passed to this method. It might be just set, or they might be loading as part of map load.

void play(long time, ActorInformation &information)

Called on each frame, with current information about player and world, in form of *ActorInformation struct*

bool interaction(std::vector<LimonAPI::ParameterRequest> &interactionInformation)

called by other entities, like Actors or Player. Used to pass information like hits, or alarming etc.

std::string getName() const

Returns the name of the Actor.

Warning: The name must be unique, or the results will be undefined.

ActorInformation struct

This struct is feeded for each frame, and meant to contain information to trigger AI behaviour. It contains the following information

Type	Name	Description
bool	canSeePlayerDirectly	Is there any object between Actor and Player.
bool	isPlayerLeft	Is Player at left of Actor.
bool	isPlayerRight	Is Player at right of Actor.
bool	isPlayerUp	Is Player higher up than Actor.
bool	isPlayerDown	Is Player lower than Actor.
bool	isPlayerFront	Is Player in front of the Actor.
bool	isPlayerBack	Is Player at back of the Actor.
float	cosineBetweenPlayer	What is the cosine of player and actor front vector.
glm::vec3	playerDirection	What is the direction vector from actor to player.
float	playerDistance	What is the distance between actor and player (unit is close to meters).
float	cosineBetweenPlayerFor-Side	cosine of the angle between right vector of actor and player.
bool	playerDead	Is player dead?
uint32_t	maximumRouteDis-tance(128)	how deep the route search should go. (maximum ~128 meters default)
std::vector<glm::vec3>	routeToRequest	Points to follow to reach the player.
bool	routeFound	Was route course successful?
bool	routeReady	Was route course done?

The first part of the information will be filled for each frame. The Route will not be filled until requested, and routeFound/routeReady will be false. To request route to player, check InformationRequest struct below.

InformationRequest struct

This struct is part of ActorInterface, and each frame Limon Engine checks all Actors for request changes. When a request is checked, its information will be reset to prevent multiple requests.

Type	Name	Description
bool	routeToPlayer	Request a route to Player
bool	routeToCustomPosition	Request a route to custom position(not implemented yet)
glm::vec3	customPosition	Position to course path

How to enable Dynamic Library discovery

Limon engine will try to load custom actors on engine startup, from libcustomTriggers file (extension based on platform). If the file is found, engine will check for a method with following signature:

```
void registerActors(std::map<std::string, ActorInterface*(*)(uint32_t, LimonAPI*)>*␣
↪actorMap)
```

This method should fill the actorMap passed, with all the custom actors, like this:

```
(*actorMap)["$ACTOR_NAME1$"] = &createActorT<$ActorClass1$>;
(*actorMap)["$ACTOR_NAME2$"] = &createActorT<$ActorClass2$>;
```

Limon Engine API Reference

uint32_t	*animateModel(uint32_t modelID, uint32_t animationID, bool looped, const std::string *soundPath*
std::string	*getModelAnimationName(uint32_t modelID)*
bool	*getModelAnimationFinished(uint32_t modelID)*
bool	*setModelAnimation(uint32_t modelID, const std::string& animationName, bool isLooped = true)*
bool	*setModelAnimationWithBlend(uint32_t modelID, const std::string& animationName, bool isLoope*
bool	*setModelAnimationSpeed(uint32_t modelID, float speed)*
std::vector<uint32_t>	*getModelChildren(uint32_t modelID)*
uint32_t	*addGuiText(const std::string &fontFilePath, uint32_t fontSize, const std::string &name, const std::*
uint32_t	*addGuiImage(const std::string &imageFilePath, const std::string &name, const glm::vec2 &positi*
bool	*updateGuiText(uint32_t guiTextID, const std::string &newText)*
uint32_t	*removeGuiElement(uint32_t guiElementID)*
uint32_t	*addObject(const std::string &modelFilePath, float modelWeight, bool physical, const glm::vec3 &*
bool	*attachObjectToObject(uint32_t objectID, uint32_t objectToAttachToID)*
bool	*setObjectTemporary(uint32_t objectID, bool temporary)*
std::vector<ParameterRequest>	*getObjectTransformation(uint32_t objectID)*
std::vector<ParameterRequest>	*getObjectTransformationMatrix(uint32_t objectID)*
bool	*setObjectTranslate(uint32_t objectID, const LimonAPI::Vec4& position)*
bool	*setObjectScale(uint32_t objectID, const LimonAPI::Vec4& scale)*
bool	*setObjectOrientation(uint32_t objectID, const LimonAPI::Vec4& orientation)*
bool	*addObjectTranslate(uint32_t objectID, const LimonAPI::Vec4& position)*
bool	*addObjectScale(uint32_t objectID, const LimonAPI::Vec4& scale)*
bool	*addObjectOrientation(uint32_t objectID, const LimonAPI::Vec4& orientation)*
bool	*removeObject(uint32_t objectID)*
bool	*removeTriggerObject(uint32_t TriggerObjectID)*
bool	*disconnectObjectFromPhysics(uint32_t modelID)*
bool	*reconnectObjectToPhysics(uint32_t modelID)*
bool	*applyForce(uint32_t modelID, const LimonAPI::Vec4 &forcePosition, const LimonAPI::Vec4 &for*
bool	*applyForceToPlayer(LimonAPI::Vec4 &forceAmount)*
bool	*attachSoundToObjectAndPlay(uint32_t objectWorldID, const std::string &soundPath)*
bool	*detachSoundFromObject(uint32_t objectWorldID)*
uint32_t	*playSound(const std::string &soundPath, const glm::vec3 &position, bool positionRelative, bool lo*
uint32_t	*getPlayerAttachedModel()*
LimonAPI::Vec4	*getPlayerAttachedModelOffset()*
bool	*setPlayerAttachedModelOffset(LimonAPI::Vec4 &newOffset)*
void	*interactWithPlayer(std::vector<ParameterRequest>& input)*
void	*killPlayer()*
bool	*interactWithAI(uint32_t AIID, std::vector<LimonAPI::ParameterRequest> &interactionInformati*
bool	*addLightTranslate(uint32_t lightID, const LimonAPI::Vec4& translate)*
bool	*setLightColor(uint32_t lightID, const LimonAPI::Vec4& color)*
bool	*loadAndSwitchWorld(const std::string& worldFileName)*
bool	*returnToWorld(const std::string& worldFileName)*
bool	*LoadAndRemove(const std::string& worldFileName)*
void	*returnPreviousWorld()*
void	*quitGame()*
std::vector<ParameterRequest>	*getResultOfTrigger(uint32_t TriggerObjectID, uint32_t TriggerCodeID)*
LimonAPI::ParameterRequest&	*getVariable(const std::string& variableName)*
std::vector<ParameterRequest>	*rayCastToCursor()*
void	*addTimedEvent(long waitTime, std::function<void(const std::vector<LimonAPI::ParameterReque*

uint32_t animateModel(uint32_t modelID, uint32_t animationID, bool looped, const std::string *soundPath)

Applies an custom animation to a model. returns model handle ID.

Parameters:

1. uint32_t modelID: handle ID of the model to animate
2. uint32_t animationID: handle ID of the animation
3. bool looped: whether the animation is looped or one off.
4. const std::string *soundPath: sound to play while animation goes. If NULL, no sound plays. Otherwise sound will be played in loop until the animation stops.

std::string getModelAnimationName(uint32_t modelID)

Returns current "Asset" animation name of the model. If a custom animation is applied to the model, it is not returned. Returns empty string when model is not found.

Parameters:

1. uint32_t modelID: handle ID of the model to check for animation name

Note: Asset Animation names are not managed by Limon, so it is possible empty string to be name of an animation.

bool getModelAnimationFinished(uint32_t modelID)

Returns true if model finished playing animation. For looped animations always returns false. Also returns false if model is not found.

Parameters:

1. uint32_t modelID: handle ID of the model to check for animation state

bool setModelAnimation(uint32_t modelID, const std::string& animationName, bool isLooped = true)

Applies an "Asset" animation to a model. Returns false if model is not found.

Parameters:

1. uint32_t modelID: handle ID of the model to animate
2. const std::string& animationName: Name of the animation to play
3. bool isLooped: Whether play animation and stop, or play in a loop

bool setModelAnimationWithBlend(uint32_t modelID, const std::string& animationName, bool isLooped = true, long blendTime = 100)

Applies an "Asset" animation to a model, blending it (using linear interpolation) with the previous animation. Returns false if model is not found.

Parameters:

1. uint32_t modelID: handle ID of the model to animate
2. const std::string& animationName: Name of the animation to play
3. bool isLooped: Whether play animation and stop, or play in a loop
4. long blendTime: How long the previous animation will effect state.

bool setModelAnimationSpeed(uint32_t modelID, float speed)

Changes animation speed by given factor. speed=2.0 will double the animation speed. Speed values < 0.001f will be rejected and return false. If model is not found it will return false

Parameters:

1. uint32_t modelID: handle ID of the model to animate
2. float speed: Animation time multiplier

std::vector<uint32_t> getModelChildren(uint32_t modelID)

Returns a vector of IDs with all children of model. Returns empty list for Model not found, as well as no children found.

Parameters:

1. uint32_t modelID: handle ID of the model to check for children

uint32_t addGuiText(const std::string &fontFilePath, uint32_t fontSize, const std::string &name, const std::string &text, const glm::vec3 &color, const glm::vec2 &position, float rotation)

Adds GUI Text to world. Returns created GUITexts handle ID.

Parameters:

1. const std::string &fontFilePath: Font file to use while rendering the text.
2. uint32_t fontSize: Font size
3. const std::string &name: Name of the GameObject GUIText
4. const std::string &text: Text to render
5. const glm::vec3 &color: Text color. Values should be between 0 and 256.
6. const glm::vec2 &position: Position of the Text. This values will be between 0 and 1. 0,0 means left bottom and 1,1 means right top
7. float rotation: Rotation of the text. 0 is upwards. it is in rads and clockwise.

uint32_t addGuiImage(const std::string &imageFilePath, const std::string &name, const glm::vec2 &position, const glm::vec3 &scale, float rotation)

Adds GUI Image to world. Returns created GUIImage handle ID.

Parameters:

1. const std::string &imageFilePath: Image files path.
2. const std::string &name: Name of the GameObject GUIImage
3. const glm::vec2 &position: Position of the Text. This values will be between 0 and 1. 0,0 means left bottom and 1,1 means right top
4. const glm::vec2 &scale: scale of the image.
5. float rotation: Rotation of the text. 0 is upwards. it is in rads and clockwise.

bool updateGuiText(uint32_t guiTextID, const std::string &newText)

Updates rendered text of the GUIText provided by the handle ID. Returns true if successful, false if handle ID invalid.

Parameters:

1. uint32_t guiTextID
2. const std::string &newText

uint32_t removeGuiElement(uint32_t guiElementID)

Removes the GUIText indicated by the handle ID. Returns 0 for success, 1 for invalid Handle ID

Parameters:

1. uint32_t guiElementID: GUIText handle ID

uint32_t addObject(const std::string &modelFilePath, float modelWeight, bool physical, const glm::vec3 &position, const glm::vec3 &scale, const glm::quat &orientation)

Adds Model to world. Returns created Model handle ID.

Parameters:

1. const std::string &modelFilePath: Model files path.
2. float modelWeight: Weight of the model. 0 means object is static, and it won't move.
3. bool physical: Whether model has physical interactions or not. If set to false, it won't collide with anything.
4. const glm::vec3 &position: World position of the Object. Please note some objects has their center set to their feet.
5. const glm::vec3 &scale: scale of the object.
6. const glm::quat &orientation: Rotation of the model.

bool attachObjectToObject(uint32_t objectID, uint32_t objectToAttachToID)

Attaches object indicated by the handle ID, to another object indicated by second parameter. Returns true for success, false for invalid Handle ID for either parameter. Attachment means if parent object move, child will move too. Example usage: bullet hole decals to dynamic objects. The object should have a transformation relative to the object it will be attached.

Parameters:

1. uint32_t objectID: handle id of the object to attach as child.

2. uint32_t objectToAttachToID: handle id of the object to attach as parent.

bool setObjectTemporary(uint32_t objectID, bool temporary)

Changes objects temporary flag. If an object is temporary, it won't be saved with map save. There is no other difference. Returns false if object can't be found. Returns true if successful.

Parameters:

1. uint32_t objectID: handle id of the object to change flag.
2. bool temporary: whether flag is set or not. True value will prevent save with the map.

std::vector<LimonAPI::ParameterRequest> getObjectTransformation(uint32_t objectID)

returns objects transformation information. If the object ID is valid, the returned vector will contain 3 vec4 parameters, translate, scale, orientation in respective order. For translate and scale, w component is not used. Orientation is in quaternion form. Returns empty vector if object not found.

Parameters:

1. uint32_t objectID: handle id of the object to get transformation.

std::vector<LimonAPI::ParameterRequest> getObjectTransformationMatrix(uint32_t objectID)

returns objects transformation matrix. If object has custom matrix generation (Physical object can define offsets), transformation might not be enough to build the matrix. This method provides objects matrix as Limon Engine has it. Returns empty vector if object not found.

Parameters:

1. uint32_t objectID: handle id of the object to get transformation matrix.

bool setObjectTranslate(uint32_t objectID, const LimonAPI::Vec4& position)

Sets objects world position to 2. parameter. Returns false if object is not found.

Parameters:

1. uint32_t objectID: handle id of the object to change position.
2. const LimonAPI::Vec4& position: new position of the object

Note: Fourth element in the vector is ignored.

bool setObjectScale(uint32_t objectID, const LimonAPI::Vec4& scale)

Sets objects scale to 2. parameter. Returns false if object is not found.

Parameters:

1. uint32_t objectID: handle id of the object to change scale.
2. const LimonAPI::Vec4& scale: new scale of the object

Note: Fourth element in the vector is ignored.

bool setObjectOrientation(uint32_t objectID, const LimonAPI::Vec4& orientation)

Sets object world orientation to 2. parameter, aka rotates it. Returns false if object is not found.

Parameters:

1. uint32_t objectID: handle id of the object to change orientation.
2. const LimonAPI::Vec4& orientation: new orientation of the object

bool addObjectTranslate(uint32_t objectID, const LimonAPI::Vec4& position)

Adds given vector to objects current world position, effectively moving it. Returns false if object is not found.

Parameters:

1. uint32_t objectID: handle id of the object to change position.
2. const LimonAPI::Vec4& position: position change desired for the object

Note: Fourth element in the vector is ignored.

bool addObjectScale(uint32_t objectID, const LimonAPI::Vec4& scale)

Scales the object, in respect to its current scale. If object is scaled to double of its original size before this call, and this call scales it to half, object will be at its original size afterwards. Returns false if object is not found.

Parameters:

1. uint32_t objectID: handle id of the object to change scale.
2. const LimonAPI::Vec4& scale: scale of object in respect to current scale.

Note: Fourth element in the vector is ignored.

bool addObjectOrientation(uint32_t objectID, const LimonAPI::Vec4& orientation)

Rotates the object from current orientation. Returns false if object ID not found.

Parameters:

1. uint32_t objectID: handle id of the object to change orientation.
2. const LimonAPI::Vec4& orientation: new position of the object

bool removeObject(uint32_t objectID)

Removes object indicated by the handle ID passed. Returns true for success, false for invalid Handle ID.

Parameters:

1. uint32_t objectID: handle id of the object to remove. Note the variable name is wrong.

bool removeTriggerObject(uint32_t TriggerObjectID)

Removes trigger volume indicated by the handle ID passed. Returns true for success, false if trigger handle ID invalid.

Parameters:

1. uint32_t TriggerObjectID: handle id of the trigger volume to remove.

bool disconnectObjectFromPhysics(uint32_t modelID)

Disconnects the model from physics, but it will be rendered as usual. Including custom and asset builtin animations. Returns true for success, false for fail. Fail can be either Handle ID invalid or the object is not a model, and can't be disconnected.

Parameters:

1. uint32_t modelID: handle id of the model to disconnect.

bool reconnectObjectToPhysics(uint32_t modelID)

Connects the model from physics. Returns true for success, false for fail. Fail can be either Handle ID invalid or the object is not a model, and can't be connected. Does nothing if already connected, returns true.

Parameters:

1. uint32_t modelID: handle id of the model to connect.

bool applyForce(uint32_t modelID, const LimonAPI::Vec4 &forcePosition, const LimonAPI::Vec4 &forceAmount)

Applies force to object using physics engine. This method have effect on only dynamic objects. Returns false if object is not found.

Parameters:

1. uint32_t modelID: handle id of the model to connect.
2. const LimonAPI::Vec4 &forcePosition: World position for the force vector to originate. Raycast results can be used for this method. Only 3 components of this method will be used, w component will be ignored.
3. const LimonAPI::Vec4 &forceAmount: Force vector. Only 3 components of this method will be used, w component will be ignored.

bool applyForceToPlayer(const LimonAPI::Vec4 &forceAmount)

Applies force to player using physics engine. Returns false if physical player is not used in this world(map).

Parameters:

1. const LimonAPI::Vec4 &forceAmount: Force vector. Only 3 components of this method will be used, w component will be ignored.

bool attachSoundToObjectAndPlay(uint32_t objectWorldID, const std::string &soundPath)

Creates a sound, attaches it to an object and plays. The sound is played in loop. Attaching an object means the sound source position and velocity will follow the object. Returns false if the object is not found.

Parameter:

1. uint32_t objectWorldID: Handle id of the object to attach.
2. const std::string &soundPath: Path of the sound to play.

bool detachSoundFromObject(uint32_t objectWorldID)

Removes the sound already attached from the object, and stops the sound. Returns false if the object is not found.

Parameter:

1. uint32_t objectWorldID: Handle id of the object to remove.

uint32_t playSound(const std::string &soundPath, const glm::vec3 &position, bool positionRelative bool looped)

Creates and plays a sound. Returns uin32_t playing sound ID.

Parameters:

1. const std::string &soundPath: Path of the sound to play.
2. const glm::vec3 &position: World position of the sound source.
3. bool positionRelative: True if position given it relative to player. Defaults to false.
4. bool looped: Play once or play in a loop. Defaults to false

uint32_t getPlayerAttachedModel()

Returns the model ID of player attachment. return 0 if player has no attachment.

Parameters:

none

Note: Player attachment might have children, check *getModelChildren method*

LimonAPI::Vec4 getPlayerAttachedModelOffset()

Returns offset of the model attached to player. returns Vec4(0,0,0,0) if player has no attachment.

Parameters:

none

bool setPlayerAttachedModelOffset(LimonAPI::Vec4 newOffset)

Sets offset to player attachment. Returns false if player has no attachment.

Parameters:

1. LimonAPI::Vec4: offset to set. w component of parameter ignored.

void interactWithPlayer(std::vector<LimonAPI::ParameterRequest> &interactionInformation)

Sends the interaction information to player Extension. If no extension is loaded, it will not have any effect.

Parameters:

1. std::vector<LimonAPI::ParameterRequest> &interactionInformation: Parameters to pass.

void killPlayer()

Kills the player.

Parameters:

none

bool interactWithAI(uint32_t AIID, std::vector<LimonAPI::ParameterRequest> &interactionInformation)

Sends the parameters to AI as new interaction. Since AI is an extension point, the parameters required are not defined by Limon engine. Returns false if no AI actor with given ID found.

Parameters:

1. uint32_t AIID: ID of AI actor to send the data
2. std::vector<LimonAPI::ParameterRequest> &interactionInformation: Parameters to pass.

bool addLightTranslate(uint32_t lightID, const LimonAPI::Vec4& translate)

Adds given translate to current position of the light indicated by the lightID. Returns false if no light with given ID found.

Parameters:

1. uint32_t lightID: ID of light to translate
2. const LimonAPI::Vec4& translate: Translate vector to add. W component will be ignored.

bool setLightColor(uint32_t lightID, const LimonAPI::Vec4& color)

Sets the color of the light, indicated by lightID parameter. Returns false if no light with given ID found.

Parameters:

1. uint32_t lightID: ID of light to change color
2. const LimonAPI::Vec4& color: RGB color to set. W component will be ignored.

bool loadAndSwitchWorld(const std::string& worldFileName)

Loads a world file, then switches the current world to the newly loaded one. If the world file was already loaded, removes the old one, effectively resetting the world. Returns false if the world file couldn't be loaded, or it is the current world. Since caller is part of current world, removing it is not possible.

Parameters:

1. const std::string& worldFileName: The file path+name of the world to load.

bool returnToWorld(const std::string& worldFileName)

Checks if the world file is loaded. If it is not, loads the world. Then changes the current world to requested one. Returns false if the world file couldn't be loaded.

Parameters:

1. const std::string& worldFileName: The file path+name of the world to load.

bool LoadAndRemove(const std::string& worldFileName)

Loads the world requested, and removes the current world. Returns true if load successful, false if not. If not successful, world doesn't change.

It is used to switch between big worlds, like game maps. It is not necessary to clear menu worlds since they use very little memory.

Note: This method clears the return previous world stack.

Parameters:

1. const std::string& worldFileName: The file path+name of the world to load.

void returnPreviousWorld()

Returns to the world that was running before current. If no world is found, it will be a noop.

Parameters:

none

void quitGame()

Clears the open devices and quits the game, shutting down the engine process.

std::vector<LimonAPI::ParameterRequest> getResultOfTrigger(uint32_t TriggerObjectID, uint32_t TriggerCodeID)

Returns the result of the trigger object. For details, check *trigger object editor*

Parameters:

1. uint32_t TriggerObjectID: The handleID of trigger object
2. uint32_t TriggerCodeID: Which triggers result is requested. 1-> first enter, 2-> enter, 3-> exit.

LimonAPI::ParameterRequest& getVariable(const std::string& variableName)

Returns variable from global variable store. If the variable is never set, it will be 0 initialized. Returned reference can be updated, doing so will be setting the parameter.

The variables are accessible by all triggers, and there are no safety checks. User is fully responsible for use of them.

Warning: The variables are not save with world itself, so they should be considered temporary.

Parameters:

1. const std::string& variableName: The name of the variable that should be returned.

std::vector<ParameterRequest> rayCastToCursor()

Returns information about what is under player cursor (crosshair). If nothing is found, empty vector is returned. if something is hit, return vector will have the following information:

1. ObjectID of the hit object
2. hit coordinates
3. hit normal
4. if Object has AI, AI id. If not, this parameter will not be in the vector.

Parameters:

none

void addTimedEvent(long waitTime, std::function<void(const std::vector<LimonAPI::ParameterRequest>&)> methodToCall, std::vector<LimonAPI::ParameterRequest> parameters)

Runs the given method, with passed parameters, after a given amount of time.

Parameters:

1. long waitTime: How long to wait before call, in milliseconds.
2. std::function<void(const std::vector<LimonAPI::ParameterRequest>&)> methodToCall: function to call.

3. std::vector<LimonAPI::ParameterRequest> parameters: parameters of that function call.

Note: Wait time is not precise beyond game ticks. Limon Engine internally ticks each 1/60 seconds.

Warning: If function is part of an object, and that object is removed, engine might crash. Avoiding those situations are game developers responsibility.

Engine Architecture

Note: This page is a work in progress. If you have any issues or questions, please open a ticket on GitHub.

Limon is a multi-platform, multi-threaded 3D game engine. Engine is built as a monolith, and architecture is not meant to be pluggable. It doesn't mean there is no extension points. The possible extension methods can be separated by 3 groups:

1. Dynamic Linking Extensions: Those are intended for game developers to use. Version 0.6 has 3 extension types:
   - Actions: These are attached to Trigger volumes, buttons or run on map load. Details can be found at *How to Implement an Action*
   - Player extensions: They attach to player, and get all inputs, as well as any requests to player interaction. *How to Implement a Player Extension* has the details.
   - AI actors: They attach to Models, and add agency to them. Details are at *How to Implement an AI Actor*
2. **Not Exposed Extensions: Those are extensions used by Engine developers.**
   - Player interface for camera and movement (ex: 3rd person support)
   - GUIObject for GUI elements (ex: Video player)
3. **Backend wrappers: Both rendering, sound, and platform(windowing, threading IO) are wrapped to their respective classe**
   - Rendering: OpenGL
   - Audio: OpenAL
   - Platform: SDL2

Note: Even though SDL2 and OpenAL are wrapped, multiple platform or sound backends not envisioned.

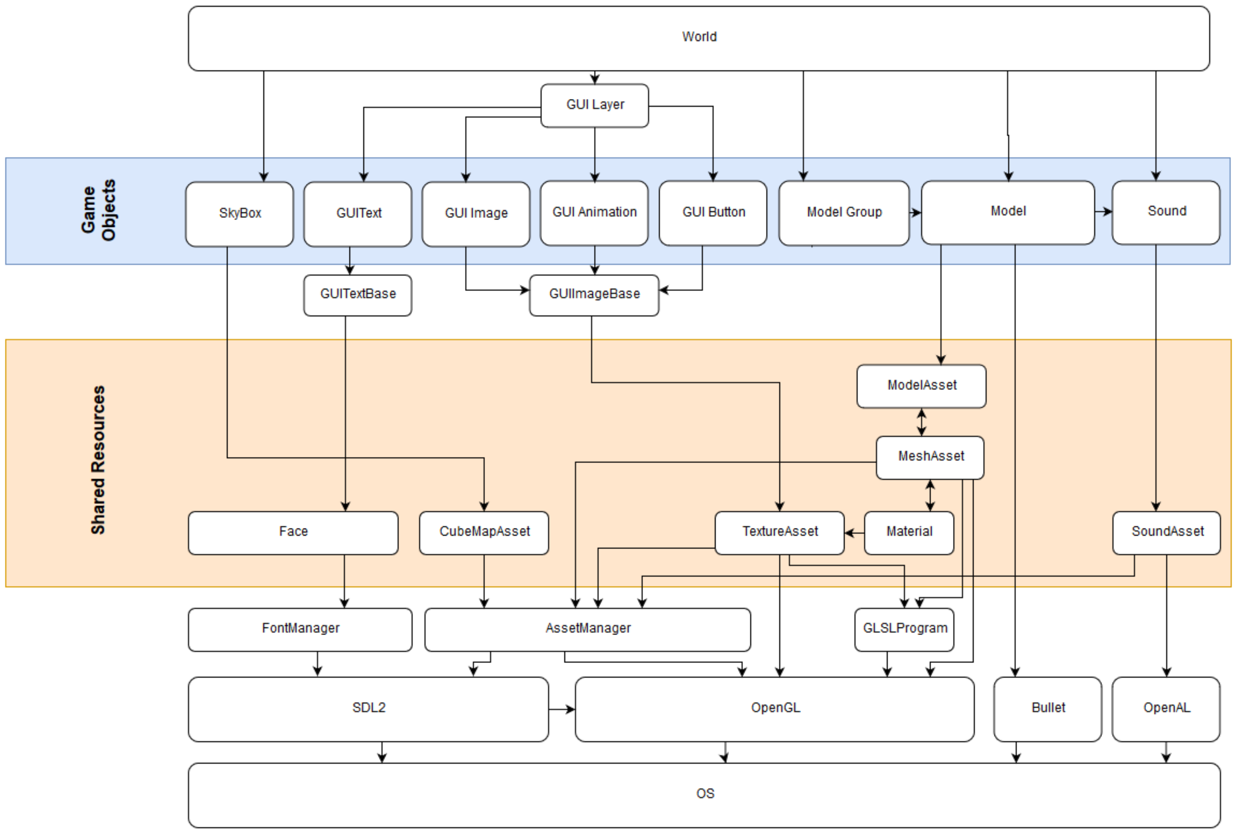
Engine Overview

Subsystems

Rendering

For rendering graphics, Limon uses a custom multi pass forward renderer, built using OpenGL.

All renderable objects are child of class *Renderable*. *World* class flags *Renderable* instances with visibility for camera and lights in simulate step.



There is a *PhysicalRenderable* child class, that is attached to the physics subsystem. Those which are child of it are also considered shadow casters, and they have *void renderWithProgram(GLSLProgram &program)* method that allows custom shaders to be used for rendering. Those objects are also batched based on their materials, and instanced rendering is used for batches, if the object is not animated. Per object world transform matrices are held in a array, and uploaded to GPU. For rendering, another array with indexes of world transforms is created and used to render the objects.

Each renderable implements *void render()* method. *World* class sets the render target, resolution etc. for shadow map generation. After that, for each light, all visible flagged *PhysicalRenderable* instances will be rendered using either *void renderWithProgram(GLSLProgram &program)* or *void renderWithProgramInstanced(std::vector<uint32_t> &modelIndices, GLSLProgram &program)* for instanced rendering. When the shadow maps for all the lights are ready, normal rendering will be done again using visiblity flags.

Physics

For physics Limon uses Bullet Physics. The only *PhysicalObject* children are registered to physics subsystem. At version 0.6, there are *Model* and *ModelGroup* classes that use Physics. The physical representation of Models are auto generated using the following logic:

1. For each mesh, if another mesh with same name prefixed with "**UCX_**" that mesh will be used.
2. Else if object is not animated, and static, use the full mesh data to generate rigid body. This is because static objects might be concave.
3. Else if object is not animated, and dynamic, auto generate a hull that encapsulates whole object. Dynamic objects are forced to be convex.

4. Else if object is animated, for each bone, auto generate a hull for the vertices that are attached to that bone. Each of those hulls animate with the object itself

Model groups just groups the models with given physical representations.

IO

IO is handled with multiple levels of abstraction. Part of platform wrapper maps raw input to engine inputs. After that, an implementer of *Player* will get the input in form of *void move(moveDirections)* and *void rotate(float xPosition, float yPosition, float xChange, float yChange)* methods. The Player class has settings that allow different types of interaction, like physical, editor etc. In the end, the inputs are feed to PlayerExtensions, so they can handle them as they see fit.

Sound

Sound backend is OpenAL. A separate thread is used to refresh sound buffers. Sound Assets and sound subsystem complexities are abstracted by Sound GameObject.

GUI

GUI is rendered and IO is handled by normal subsystems. GUI objects must implement *GUIRenderable* class. If the current *Player* is flagged with *menuInteraction*, the GUI objects will react to player input.

Editor

Editor is built using Dear ImGui library. Main part of it is within *World* class, but to enable custom behaviour, objects get to implement their own editor interfaces.

Game Play and Game Objects

As of version 0.6, Limon engine has following game objects:

- Player
- Light
- Model
- Model Group
- SkyBox
- Trigger Object
- Gui Text
- Gui Image
- Gui Button
- Gui Animation
- Sound

Those object can be used in Editor, and by Triggers. Gameplay layer has an API called LimonAPI, and it has an interface to allow extending, and Limon Engine supports dynamically loading those custom triggers. For details, please check *How to Implement an Action*

AI

Limon has an interface called *ActorInterface* that is used to allow custom AI implementations to be used. Each actor will be triggered each simulation step with *ActorInformation*, which contains the player direction, whether or not player is visible etc. It is possible to ask for a route to player using this interface too, assuming actor is same size with the player.

Road Map

Latest release of the Limon Engine is 0.6. This page is meant to indicate the development plan up until 1.0. Please note information in this page is subject to change, and should not be considered final.

Version 0.6

Main goal of this release was fighting. All of the listed functionality implemented.

1. Asset discovery should be automatic, instead of current asset list approach.
2. Allow attaching models to player. Should be used to carry weapons around.
3. Allow listening for input by triggers. Should be used for attack
4. Provide what is under the cursor, with its distance to player. Should allow checking for whether player hit something or not
5. Provide "distance to player" to Actors, so AI can determine melee attack.
6. Allow trigger <-> Actor communication for hits.
7. Allow adding quad/s as bullet holes etc.
8. There is no way to implement Player getting hit. A way should be exposed.

Version 0.7

1. Material editor.
2. Particle emitters.
3. Shadow mapping improvements.

Before 1.0

1. Generate AI navigation mesh from AI navigation grid. Serialize it with map. Done
2. Make world a tree, and allow modifications to groups. Done
3. Options can't be set using GUI, they should have.
4. Mixamo support. Done
5. Directory listings should auto generate for assets, and it should be able to refresh. Partially done, no refresh.

6. There is no stair support, there should be.
7. Editor should support undo/redo.
8. Auto Align objects.

Possible Additions

1. Custom shaders
2. Vulkan backend
3. Python scripting support
4. Android support
5. Emscripten/webassembly support

Authors

Core developers

- Engin Manap

Special Thanks

- Alper Tekinalp
- Mesutcan Kurt

CHAPTER 2

Indices and tables

- genindex
- modindex
- search