

SqrMelon

A tool for keyframe animation & fragment shader management for 64k executables.



Even the logo is made with it!

This document was last updated 23 April 2018, we intend the UI & setup documentation to remain relevant, but limit technical documentation as much as possible as it will rapidly become outdated.

Download it at:

<https://github.com/trevorvanhoof/sqrmelon>

Setup guide

1. Download & run with default settings:

<https://www.python.org/ftp/python/2.7.12/python-2.7.12.amd64.msi>

2. Download & run with default settings:

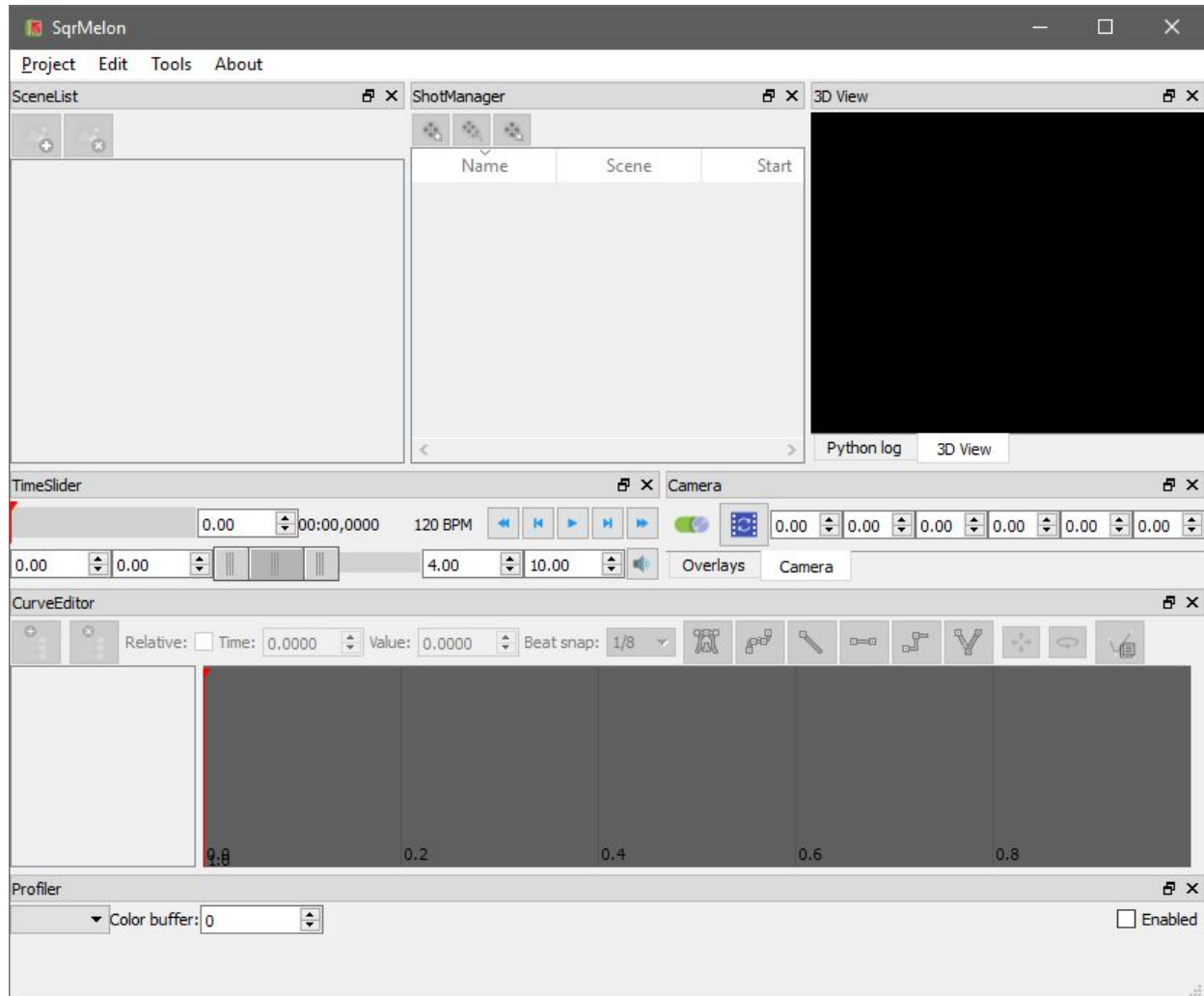
<https://sourceforge.net/projects/pyqt/files/PyQt4/PyQt-4.11.4/PyQt4-4.11.4-gpl-Py2.7-Qt4.8.7-x64.exe/download>

3. Run install.bat

[optional mp3 support] 4. Download & run with default settings:
<https://github.com/downloads/AVbin/AVbin/AVbin10-win64.exe>

5. Run SqrMelon.exe (it should automatically run as administrator)

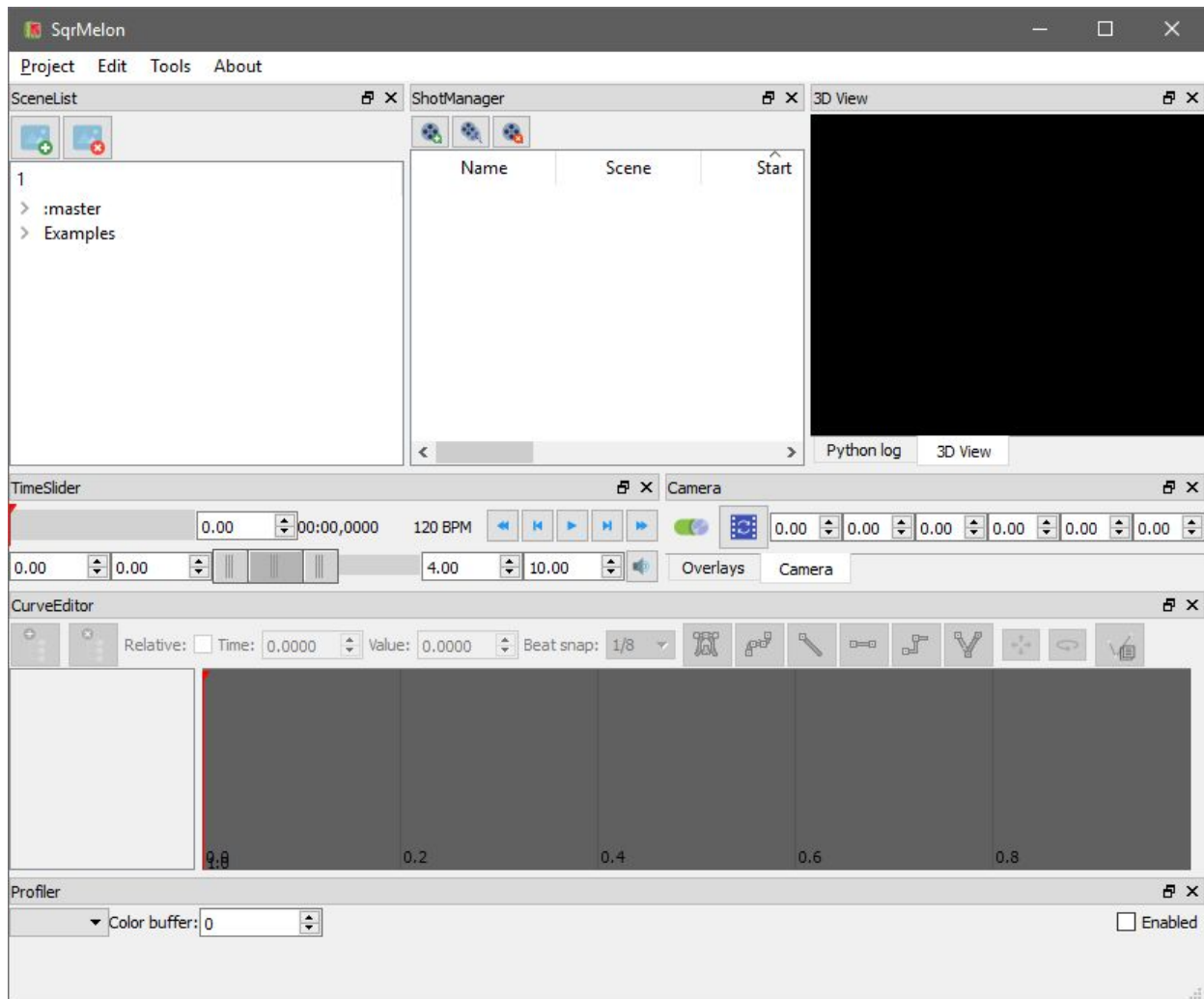
And that should pop up something like this:



Getting started & UI guide

In this section I will go over all the UI elements in an order that makes sense when you are trying to create your first project.

In the menu bar, you can simply create a new project. This asks you to save it somewhere, it will create a new folder with that name and copy over a default project.



Creating a scene

On the left you will see the SceneList has been enabled, if you click this icon:



You can create a new scene and give it a name.

Scenes are what contain shader code, based on the template pieces of the shader graph are exposed and can be altered per-scene, such as the distance function, materials and lighting functions.



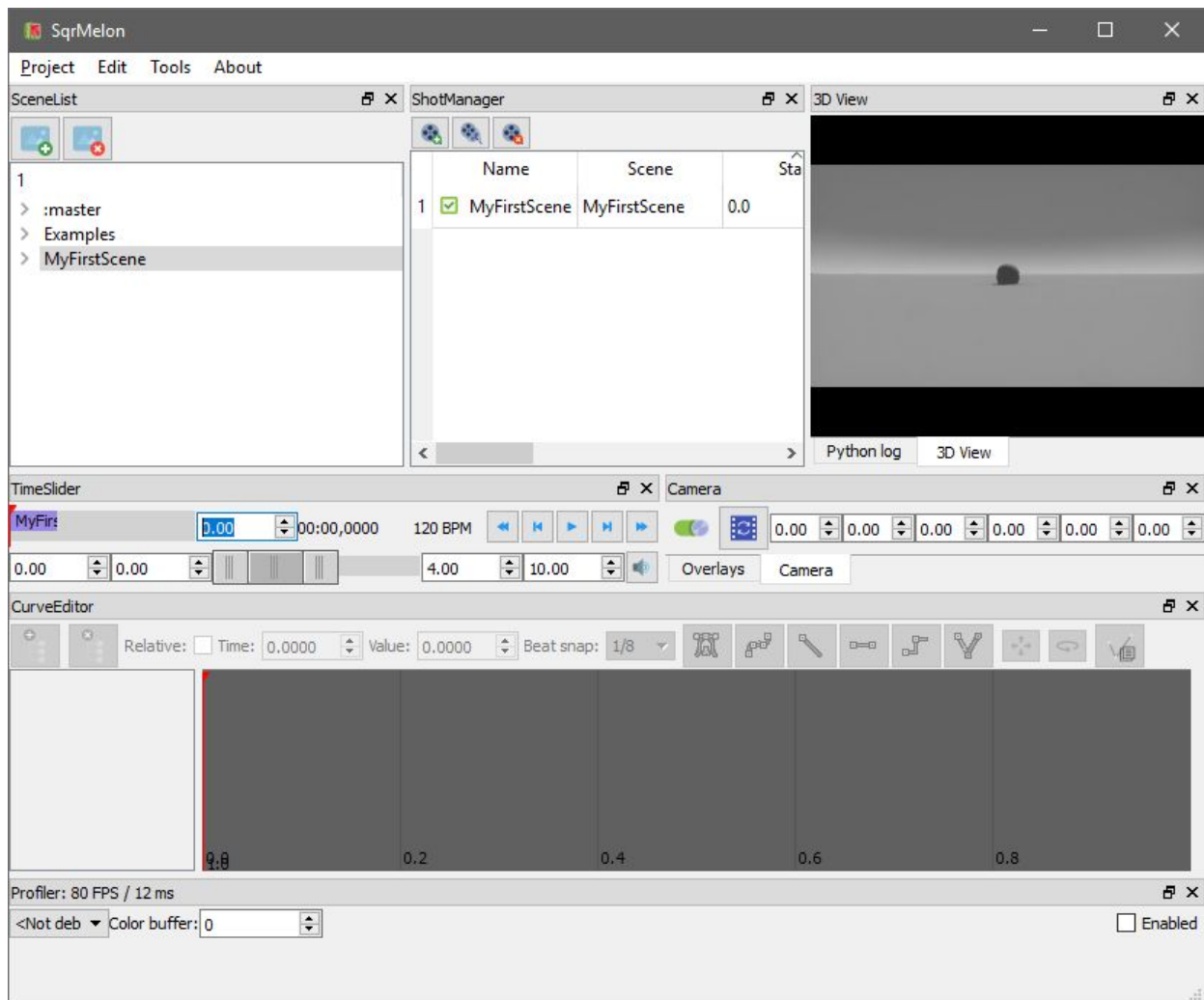
Creating a shot

Next you can create a new shot with the ShotManager by clicking:



This will show you a basic template scene in your viewport.

Troubleshooting: If the screen remains black, try to click and drag in the TimeSlider to force the camera to update.



Shots are what contain animation data, and are the thing you “save” when you hit **CTRL + S** (or use the Project menu at the top menu bar).

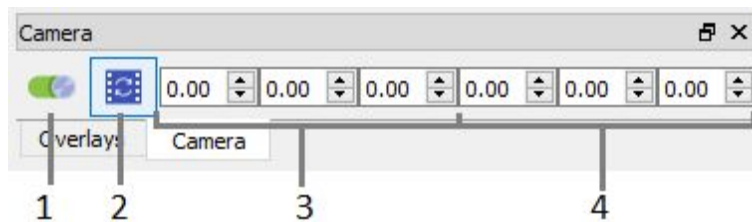


Navigation in 3D

The 3D view supports keyboard and mouse input. Here is the key map:

W & S	Move forwards and backwards
A & D	Strafe left and right
Q & E	Strafe up and down
Arrow keys	Rotate in direction of key
Home & End	Roll CCW & CW
Click & Drag	Horizontal for heading, vertical for pitch

Altering your camera position will be reflected in the “Camera” control. Here you can also:



1. Turn off animation of the camera
2. Snap the camera to it's animated position
3. Enter translation XYZ
4. Enter rotation XYZ (in degrees)

Controlling animation playback

In the shot manager you can double click values, for example double clicking duration and turning it into 4 makes the shot take 4 beats (or 1 musical bar if your music is 4/4).

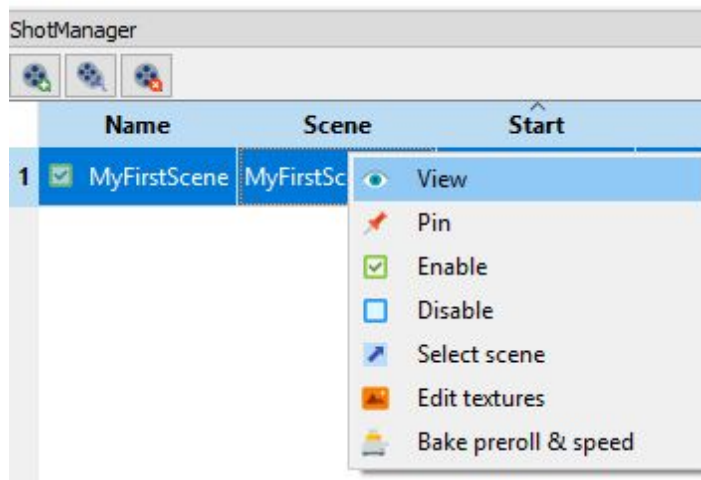
A screenshot of the 'ShotManager' window. It has a title bar with a lock icon and a close button. Below the title bar are three icons: a play button, a stop button, and a camera button. The main area is a table with the following columns: 'Name', 'Scene', 'Start', 'End', and 'Duration'. The first row of data has the following values: '1' in the first column, a checked checkbox in the second, 'MyFirstShot' in the third, 'MyFirstScene' in the fourth, '0.0' in the fifth, '4.0' in the sixth, and '4.0' in the seventh. The '4.0' in the seventh column is highlighted with a blue selection box.

ShotManager						
	Name	Scene	Start	End	Duration	
1	<input checked="" type="checkbox"/> MyFirstShot	MyFirstScene	0.0	4.0	4.0	

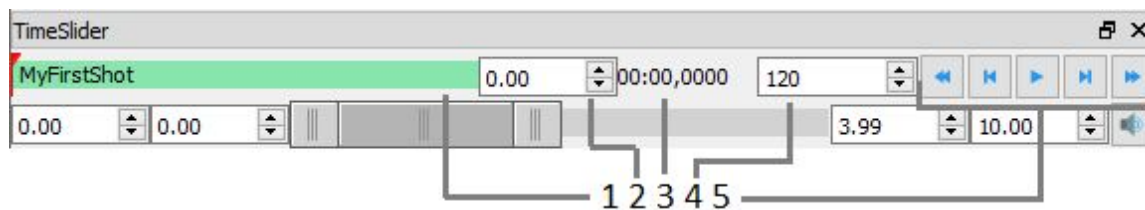
It is important to know that all time related values are handled in BEATS, so changing the BPM scales the playback speed of the entire demo.



Right clicking a shot reveals a “view” option, this option automatically sets the time slider to loop that one shot.



The time slider is essentially a track showing the active shots with some playback control buttons.



The top half shows:

1. Currently looped time range / shot layout, click and drag to move play head (red thing on the left).
2. Current play head position in BPM.
3. Current timecode (converted from current play head using the BPM).
4. BPM, double click to edit, saved with the project data.
5. Playback controls, jump to start, step -1 beat, play, step +1 beat, jump to end.



The bottom half controls loop ranges:

1. Demo start
2. Loop start
3. Loop end
4. Demo end
5. Mute / unmute audio playback (first mp3 or wav file in project folder)
6. Slider to move looped time (snapped to whole beats)

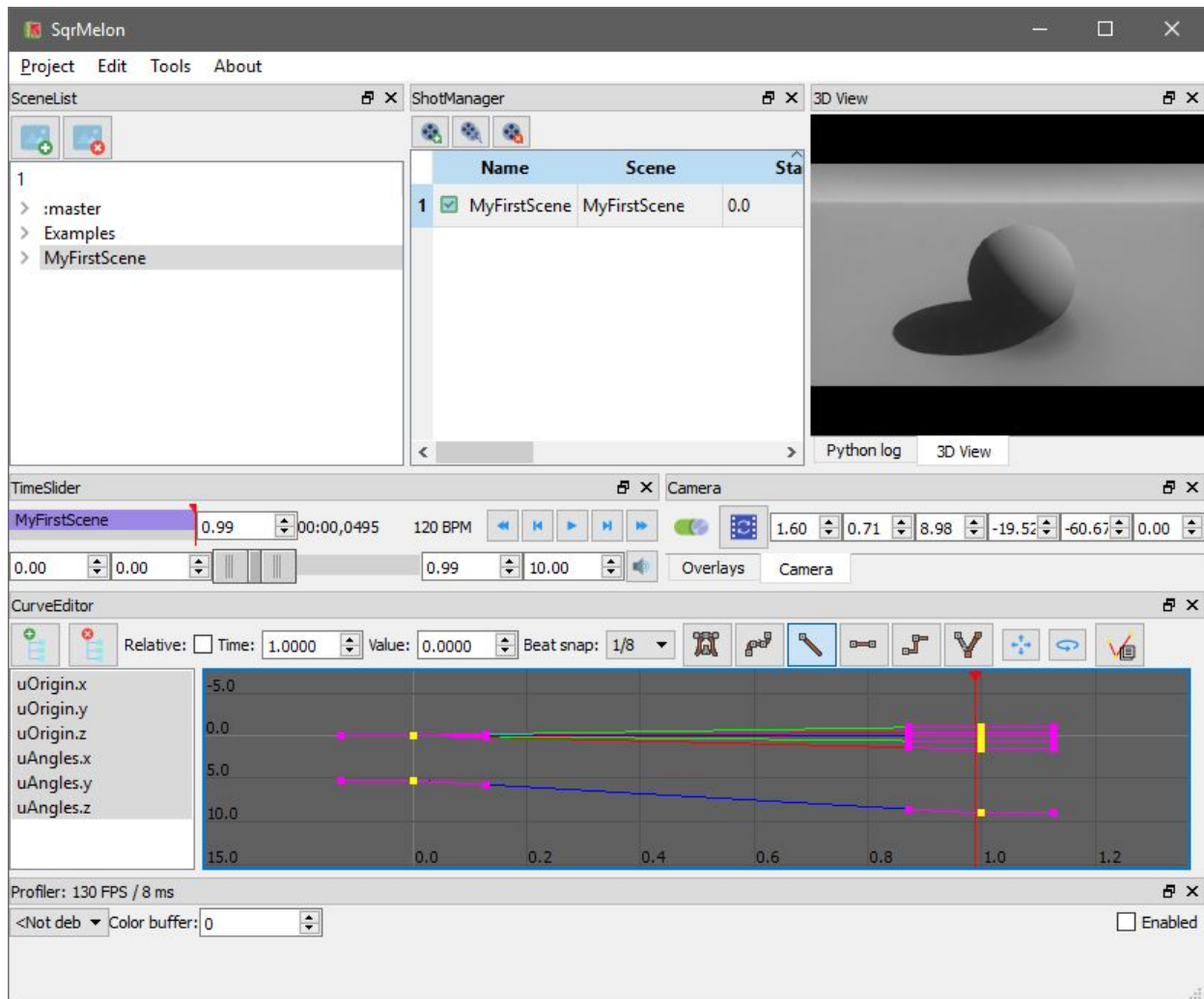
Keyframes

If you click on the shot in the shot manager, the CurveEditor will become active. Animation channels for camera position and rotation are always automatically created, so you may see a list of inputs already.

Pressing “K” will set a “Camera Key” at the current time, based on the current camera position in the 3D view.

With the right click -> view option in the shot manager it is easy to control animation at the start and end. This is the essential workflow to create a linear camera animation:

1. Right click -> view shot from the shot manager
2. Move the play head to the start of the shot (jump to start button the time slider).
3. Move the camera in the 3D viewport.
4. Press “K” to set a key.
5. Move the play head to the end of the shot (jump to end button in the time slider).
6. Repeat steps 3 & 4.
7. Select all the keys in the curve editor (Press “A” to automatically frame the camera).
8. Click this button to make the tangents linear:



Sorry if that went a little fast, this is probably a good time to create some shots and get the hang of the camera and animation controls.

A more detailed description of the curve editor and its many possibilities is found in the "UI & Hotkey Reference".

Shader editing

Simply expand a scene and double click any shader file name in the SceneList to open the corresponding .glsl file in your default text editor. Saving the file will hotload it automatically.

In the SceneList there are 2 types of shader groups. Scenes that you create will expand to editable shader files inside of them. The templates of the project may also expose shared shader code (used by all scenes derived from this template). These will be in a group with the template name, prefixed by a colon.

- Use the OS to set the right editor for .glsl, .vert and .frag files.
- What you can do in the editable code is very template specific. More on that in the template guide.



Animating on the beat

Besides keyframes all being on the beat, it is possible to use the uniform “uBeats” to get a (timeline-global) cursor in beats. This can be used to e.g. pulsate materials on the beat:

```
step(0.5, fract(uBeats));
```

Adding and using a custom animation uniform

It is possible to define additional uniforms anywhere in the shader code, either shared or scene-specific. After adding (and visualizing) a uniform it can immediately be animated from the graph editor. Simply select a Shot in the ShotManager, then go to the CurveEditor and click this button to add a new animation channel:



Here are some naming conventions:

The name must match the name of the uniform you want to animate.

Suffix a channel with [xy], [xyz], or [xyzw] to animate a vec4.

This will automatically create the right channels, animated separately.

UI & Hotkey reference

Cheat sheet of buttons, hotkeys and other interactions per UI element.

Menu bar

Project menu

1. New

Select a directory to create a new project in (will create new directory at destination folder).

2. Open

Open a project (.p64 file).

3. Save (**Ctrl + S**)

Save all shots & animation changes done to the project.

Note that shader changes are already saved on disc.

Edit menu

1. Undo (**Ctrl + Z**)

Undo animation data change.

Note that it does not switch the view, so you may be undoing curves in a shot that you're not looking at.

2. Redo (**Ctrl + Y**)

Redo animation data change.

Note that it does not switch the view, so you may be redoing curve changes in a shot that you're not looking at.

3. Undo view change (**I**)

Undoes CurveEditor view change.

4. Redo view change (**J**)

Redoes CurveEditor view change.

5. Key camera (**K**)

Key current camera position and rotation, at the current time, into the shot selected in the ShotManager. Ignored if no active shot.

6. Toggle camera control (**T**)

Switch between animation playback and free camera, which will retain custom camera position when pressing play.

7. Snap camera to animation (**R**)

Move free camera to match animation data at the current time.

Tools

1. Color Picker

Opens a color picker, after clicking OK a string of GLSL code will be in the clipboard, ready to be used in a shader.

2. Lock UI

Disable docking, undocking and resizing of components. Floating components are not locked.

3. Full screen viewport (**F11**)

Assuming the viewport is floating, it will become borderless full-screen on the display it's currently on, without resizing the actual render resolution, allowing you to set the viewport at 720p and then pressing F11 to get a scaled 720p output on any resolution...

4. Preview resolution

This sub-menu shows some hard coded resolutions, click it to force-resize the viewport (window) to this size.

Also there are options to render at “½ view” and “¼ view”. This simply resizes the size of OpenGL render targets but upscales it back to fill the viewport at the current size. We use it for getting decent framerates on old laptops.

5. Record

Takes the current playback range and does a frame-by-frame capture to JPG files, inside a Capture/ folder next to the application exe. Resolution and framerate options will pop up.

It also generates a “**convertcapture**” folder with a batch file to turn the images to an mp4 (convert.bat) or gif (convertGif.bat). This requires **ffmpeg**. You probably want to 64-bit version from: <https://ffmpeg.zeranoe.com/builds/>.

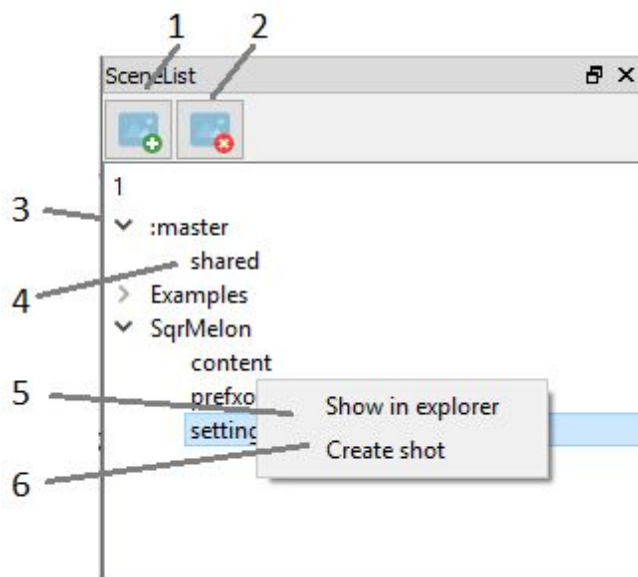
Inside which you’ll find “<something>-win64-staticbin/ffmpeg.exe” which must be placed inside the “convertcapture” folder.

If you have audio in your project you will also see “**merge.bat**” which will take the merged mp4 and the soundtrack of your project (if any) and turn it into “**output.mp4**” for a high quality capture with audio. *Note: this will only work after having played audio since tool launch.*

About

Simple about dialog showing where the tool came from, and some credits.

SceneList



1. Add scene

Prompt for a filename and create a new scene. This will create a new set of xml & shader files on disk, representing the shot & scene data.

2. Delete scene

Delete the xml and shader files from disk (and update the SceneList). This also deletes all shots associated with the scene.

3. Expand / collapse template & scene

Every scene listed can be expanded to show relevant shader files.

4. Double click shader file

Double click a shader file in the list to open it in an (OS default) editor.

5. Context menu: Show in explorer

Right click a shader file and open windows explorer at its location.

6. Context menu: Create shot

See ShotManager: Create shot. Selects the right scene automatically.

ShotManager

The screenshot shows the ShotManager window. At the top is a toolbar with three icons: a green plus (1), a blue magnifying glass (2), and a red X (3). Below the toolbar is a table with columns: Name, Scene, Start, End, Duration, Speed, and Preroll. The first row (6) is highlighted in blue and contains the values: 1, Test, Test, 0.0, 8.0, 8.0, 1.0, 0.0. Red vertical lines with numbers point to specific elements: 4 points to the Start column header, 5 points to the End column header, 7 points to the Start value (0.0), 8 points to the End value (8.0), 9 points to the Duration value (8.0), 10 points to the Speed value (1.0), and 11 points to the Preroll value (0.0).

	Name	Scene	Start	End	Duration	Speed	Preroll
1	Test	Test	0.0	8.0	8.0	1.0	0.0

1. Create shot

Add a shot, pops up to set a name and source scene.

2. Duplicate shots

Duplicate all selected shots.

3. Delete shots

Delete all selected shots

4. Sort

Click column names to sort. It is recommended to sort by Start to view shots in chronological order.

5. Select shot

Click to select a shot

Hold Ctrl to toggle

Hold Shift to select a range of shots

6. Rename

Double click to change the name of a shot.

7. Start

Double click to change start beat (end is adjusted accordingly).

8. End

Double click to change end beat.

9. Duration

Double click to change duration beat (end is adjusted accordingly).

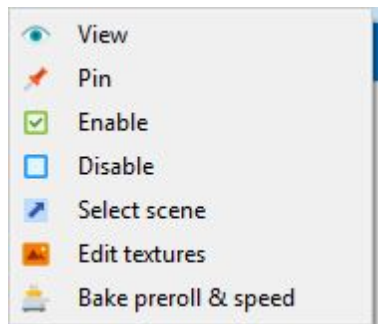
10. Speed*

Double click to change animation speed multiplier (useful to slow down or speed up existing animations).

11. Preroll*

Double click to change animation data offset in beats (after speed is applied).

* Speed and preroll are not supported in the demo player code. Use "Bake preroll & speed" in the context menu to adjust animation data once values are final.



Context menu

View

Set the timeline loop range to match this shot's start & end.

Pin

Force this shot to always render, regardless of the play head position.

Enable (Also functions as UnPin)

Use this shot on the timeline.

Disable

Do not consider this shot as part of the content.

Select scene

Highlight the scene in the SceneList (for quick shader selection).

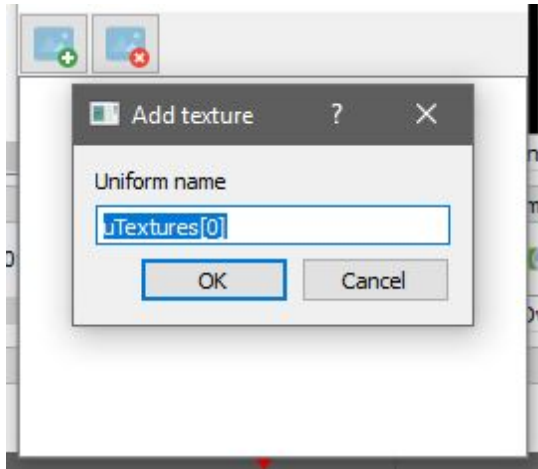
Edit textures

More below

Bake preroll & speed

Bakes preroll and speed settings, these settings are not supported in the demo player code so this helps you adjust animation to match what you see.

Edit textures



In this dialog you can add and remove user images and bind them to a specific uniform. These settings are unique per shot.

Textures are not supported in the demo player code, but can be used for mockups or to bake down data that is otherwise generated in custom player code.

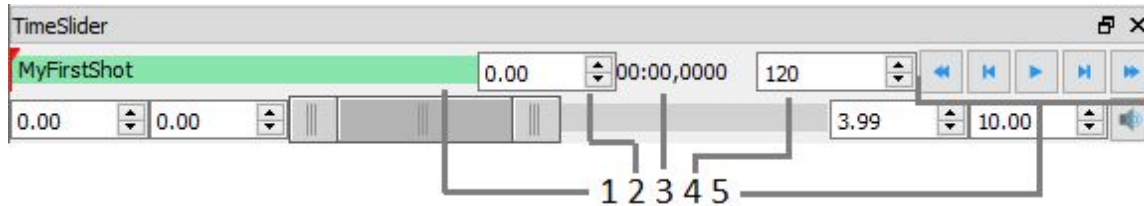


Viewport

The 3D viewport supports free camera movement when not playing or when disabling camera animation (T).

W & S	Move forwards and backwards
A & D	Strafe left and right
Q & E	Strafe up and down
Arrow keys	Rotate in direction of key
Home & End	Roll CCW & CW
Click & Drag	Horizontal for heading, vertical for pitch
Shift (hold)	Faster keyboard input
Ctrl (hold)	Slower keyboard input

TimeSlider



The top half shows:

1. Currently looped time range / shot layout, click and drag to move play head (red thing on the left).
2. Current play head position in BPM.
3. Current timecode (converted from current play head using the BPM).
4. BPM, double click to edit, saved with the project data.
5. Playback controls, jump to start, step -1 beat, play, step +1 beat, jump to end.

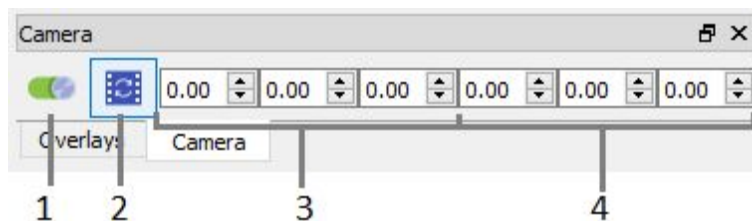
Pro tip: Ctrl+Click in the timeline to automatically select the shot in the ShotManager & activate its animation in the CurveEditor.



The bottom half controls loop ranges:

1. Demo start
2. Loop start
3. Loop end
4. Demo end
5. Mute / unmute audio playback (first mp3 or wav file in project folder)
6. Slider to move looped time (snapped to whole beats)

Camera



1. Turn off animation of the camera
2. Snap the camera to it's animated position
3. Enter translation XYZ
4. Enter rotation XYZ (in degrees)

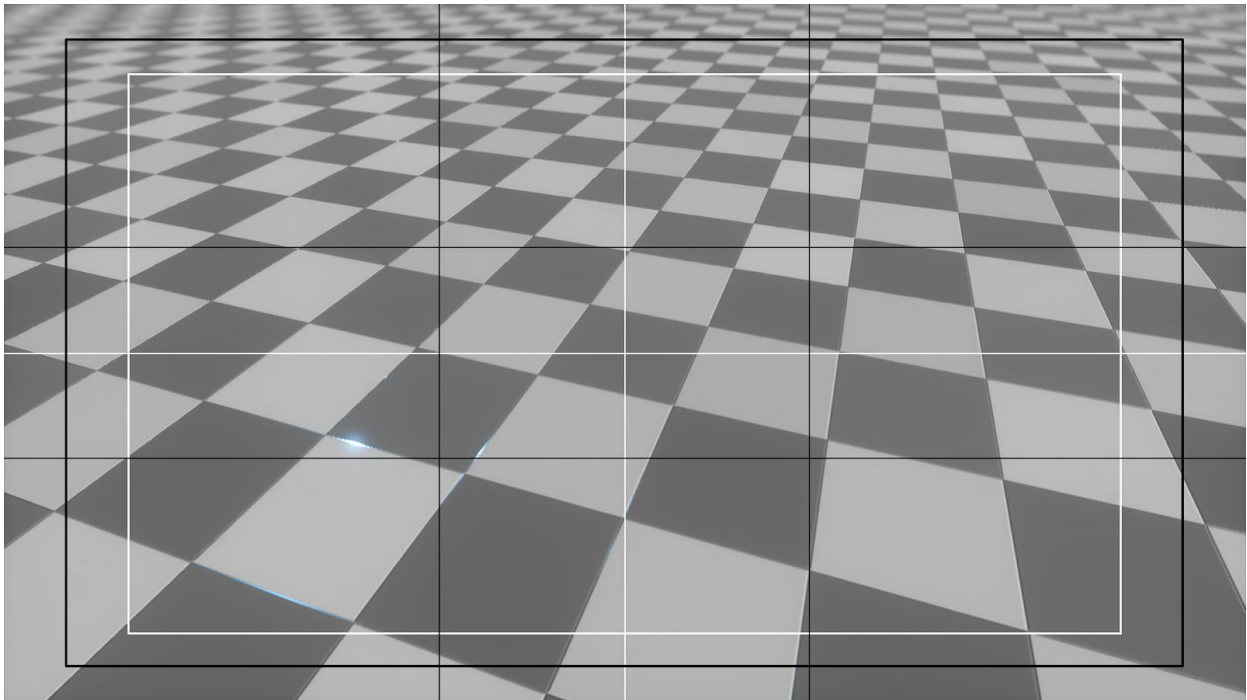
Overlays



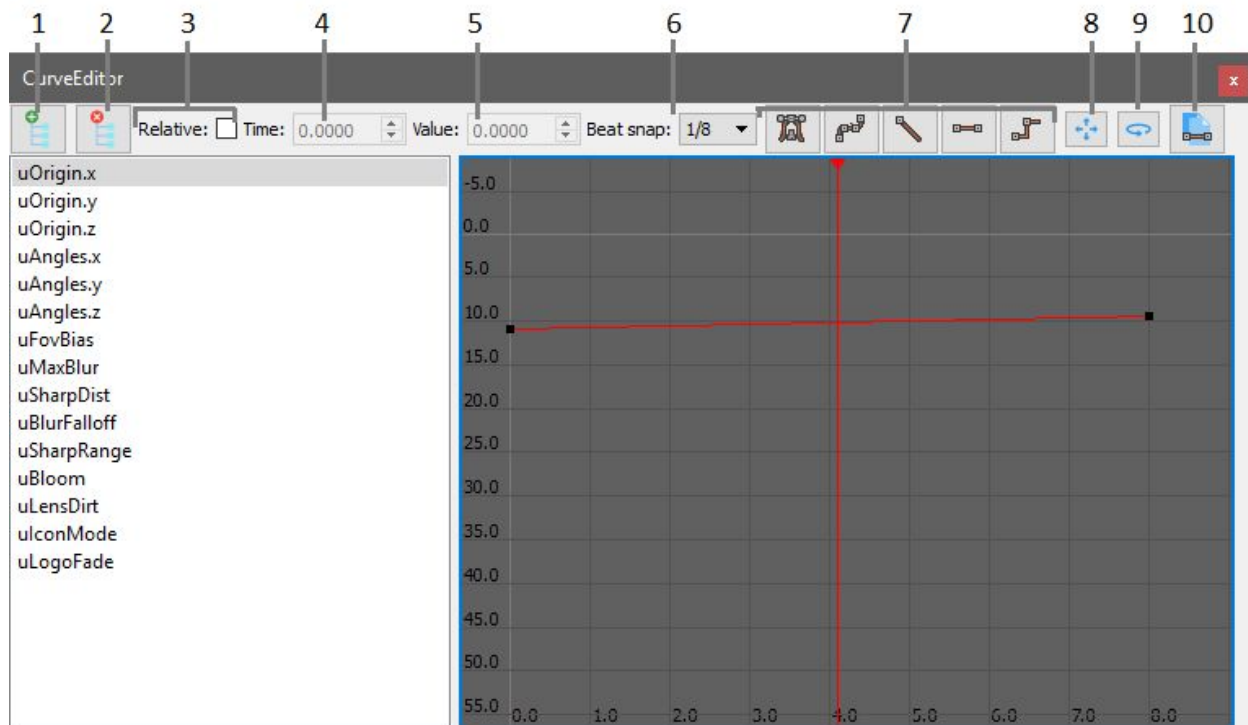
Manages debug overlays drawn on top of the rendered result. Overlays like these may help checking if important actions are well framed on the screen & avoid action near the edges (easily missed by an audience in front of a beamer).

Click on the colored square to change the color (multiplier).

Add more (transparent) PNG files to the overlays/ folder in the application directory as you like, they are detected automatically.



CurveEditor



1. Add channel

Specify the name of the uniform to animate.

If the uniform is a vec2, vec3 or vec4, specify the name suffixed with one of:

[xy] [xyz] [xyzw]

For example: uOrigin[xyz] to animate the camera position (vec3).

Talking to a uniform with animation of the wrong dimension throws errors in the Python Log. Delete the channels and recreate them if this is the case.

2. Remove selected channels

Deletes all selected channels. **Not undoable!**

3. Relative input

Values entered at Time & Value are applied additively to the selected keys.

4. Time

This shows the current time of the selected key(s), if keys have different times this is simply the first selected key. Editing this value applies the same time to all selected keys.

5. Value

This shows the current value of the selected key(s), if keys have different values this is simply the first selected key. Editing this value applies the same value to all selected keys.



Beware of using tab, tabbing into the Value input and then losing focus moves all selected keys to the same value (e.g. 0).

6. Beat snap

The time value is always snapped to the beat. When setting, inserting or moving keys around the beat snap is the precision with which the values will snap.

7. Tangent modes

If you're not a keyframe animator (used to Maya) this may make little sense. These buttons change the shape of the curve between selected keys.

These are guidelines that will make your animations seem to go at a constant velocity. I highly recommend using *linear* for all start and end keys:  and to refrain from additional keys unless necessary, in which case they should use *spline*: . If you know what you're doing you can of course pull off ease-in/out and other things, but especially cameras will look jarring if they don't calmly accelerate and steer.

8. Key camera translation into selection

Select 1-3 channels (order matters!) and click this button to apply the current camera's position X, Y, Z to these channels. This allows you to fly up to a point and key another uniform there, to more easily position other objects in 3D space.

9. Key camera angles into selection

Select 1-3 channels (order matters!) and click this button to apply the current camera's rotate X, Y, Z to these channels.

10. Duplicate keys

This re-inserts the selected keys at the current time.

Channel list

Click to select a channel

Hold Ctrl to toggle channels

Hold Shift to select a range of channels

Context menu:

1. Copy selected channel(s)

Puts curve data in the clipboard

2. Paste channels

Paste from the clipboard (to e.g. transpose channels to another shot)

3. Paste into selected channel

Works for only 1 channel at a time, allows copying the curve data from one channel to another.

Curve area

Shows all channels selected in the channel list & allows manipulation of curve data.

If you're a Maya user you'll find a lot of similarities.

Camera

Ctrl + A selects all channels

A frames the camera on all visible keys

F frames the camera on all selected keys

Alt + click & drag to pan

Alt + right click & drag to zoom in and out

Hold shift to zoom in only 1 axis, vertical or horizontal is based on initial mouse movement

Keys

Click & drag in empty space to select keys

Hold shift to toggle selection instead of selecting something completely new

Ctrl + click & drag to move the play head

Click & drag a selected key to move all selected keys

Hold shift to move in only 1 axis, vertical or horizontal is based on initial mouse movement

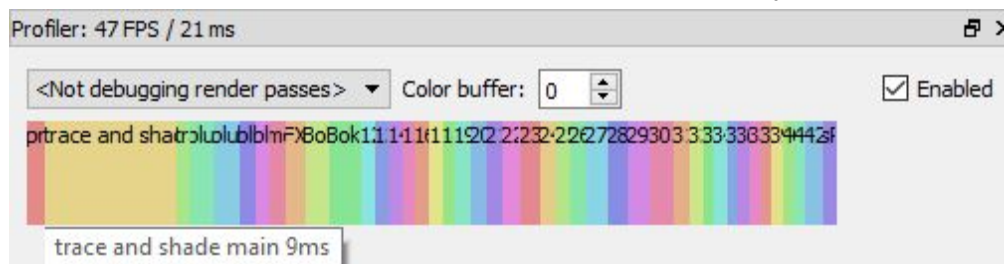
I inserts a new key at the play head for all visible curves.

This may break tangent continuity.

K inserts a new key on uOrigin[xyz] and uAngles[xyz] with the current camera position & rotation.

Profiler

The title bar shows the framerate average of 10 frames. Playback for best result.



The enabled check box in the top right can be used to profile in more detail, this will slow down playback but in return render a detailed set of bars, size representing duration of a particular render pass.

Tooltips show pass name and time spent on it during the last frame.

The drop down and number labeled “Color buffer” are to debug render buffers of the template. It lists all draw calls and will stop drawing and present to the screen at that point if used. This should not be used simultaneously with the profiler enabled.

PythonLog

A simple reroute of any python errors. Since the application is mostly written in python, it most likely won't crash when an error occurs, but just fail to behave. In this window you can review any output and exceptions raised in the background.

Template creation guide

This is kind of a hidden part of the tool, intended to be used by people developing a core render pipeline to be used by others. From here we can hide and expose code that we want others to touch, so that content creators can't easily break things and simply focus on scene content and animation.

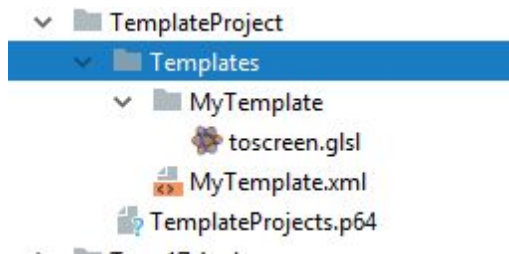
Note that the player code only supports projects with 1 template, but the tool can create scenes from multiple templates.



Inside the Templates/ folder of a project the templates reside. A template is an XML file that describes where to find shader components and how to combine them into passes. They also describe render buffer usage and resolution. It is up to the user to reduce render buffer count as much as possible, to save on VRAM. The template XML looks for shaders inside a subfolder with the same name, but paths are relative so it is possible to navigate elsewhere with “../”.

A template that draws to the screen

Let's create a new template, “MyTemplate”. This is the project structure:



The .p64 file is blank.

toscreen.glsl renders a gradient based on the pixels & resolution:

```
#version 410
```

```

uniform vec2 uResolution;

out vec4 outColor0;

void main()
{
    outColor0 = vec4(gl_FragCoord.xy / uResolution, 0, 0);
}

```

MyTemplate.xml contains a template that uses this file to draw to the screen:

```

<template>
  <pass name="draw to screen">
    <global path="toscreen.glsl"/>
  </pass>
</template>

```

A pass represents a call to glRect, with the listed glsl files concatenated into a single fragment program. The name is shown in the profiler UI to make browsing through passes easier.



Static textures

Next, I will add a preceding pass that draws some lines.

lines.glsl:

```

#version 410

uniform vec2 uResolution;

out vec4 outColor0;

void main()
{
    vec2 uv = gl_FragCoord.xy / uResolution;
    float p = uv.x + abs(uv.y - 0.5);
    float lines = smoothstep(-0.1, 0.1, sin(p * 3.14159265359 * 4.0));
    outColor0 = vec4(lines);
}

```

MyTemplate.xml:

```
<template>
  <pass name="lines">
    <global path="lines.glsl"/>
  </pass>
  <pass name="draw to screen">
    <global path="toscreen.glsl"/>
  </pass>
</template>
```

This is rendering to screen twice, so the second pass overwrites the first. To combine passes, we can render into **buffers** and forward these buffers **inputs** into another pass:

toscreen.glsl

```
#version 410

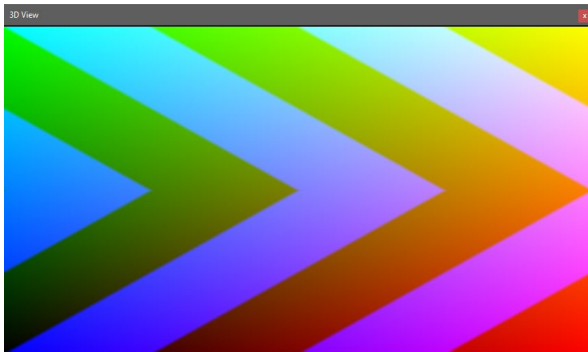
uniform vec2 uResolution;
uniform sampler2D uImages[1];

out vec4 outColor0;

void main()
{
  vec4 inputColor = texelFetch(uImages[0], ivec2(gl_FragCoord.xy), 0);
  outColor0 = vec4(gl_FragCoord.xy / uResolution, inputColor.x, 0);
}
```

MyTemplate.xml:

```
<template>
  <pass buffer="0" name="lines">
    <global path="lines.glsl"/>
  </pass>
  <pass name="draw to screen" input0="0">
    <global path="toscreen.glsl"/>
  </pass>
</template>
```



The number in input0 is the index representing the texture in the shader. This is automatically forwarded to the "ulimages" uniform array. The value of this attribute should match the value of a buffer attribute on a preceding pass.

Now this input may be completely static, and far more expensive to calculate than this simple pattern. In this case you can use the "static" attribute, to make sure a pass is computed only once (during demo playback, the tool will still hotload GLSL changes).

MyTemplate.xml:

```
<template>
  <pass buffer="0" name="lines" static="1">
    <global path="lines.glsl"/>
  </pass>
  <pass name="draw to screen" input0="0">
    <global path="toscreen.glsl"/>
  </pass>
</template>
```

Because it's weird to have aspect ratio in your texture, we can conveniently add a size attribute to create a square buffer of the given size.

MyTemplate.xml:

```
<template>
  <pass buffer="0" name="lines" static="1" size="128">
    <global path="lines.glsl"/>
  </pass>
  <pass name="draw to screen" input0="0">
    <global path="toscreen.glsl"/>
  </pass>
</template>
```



This will change our visual a little, due to our usage of texelFetch. If instead we calculate 0-1 UV coordinates and fetch those, our texture will scale and tile!

toscreen.glsl:

```
#version 410
```

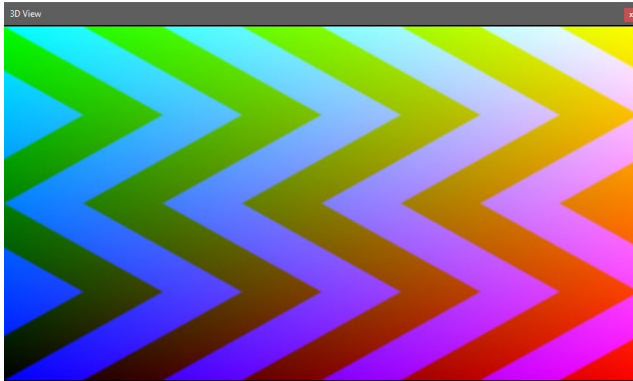
```

uniform vec2 uResolution;
uniform sampler2D uImages[1];

out vec4 outColor0;

void main()
{
    vec4 inputColor = texture(uImages[0], (gl_FragCoord.xy / uResolution) * 2.0);
    outColor0 = vec4(gl_FragCoord.xy / uResolution, inputColor.x, 0);
}

```



Low-resolution passes

We can also create buffers at a factor of the screen size.

Note that you should not reuse buffers of different sizes.

For demonstrative purposes I've added an animated dot pass, that then gets downsampled to a half resolution buffer.



The result no longer looks crisp as it is upscaled from half resolution, but that means it works!

MyTemplate.xml:

```

<template>
  <pass buffer="0" name="lines" static="1" size="128">
    <global path="lines.glsl"/>
  </pass>
</template>

```



```

</pass>
<pass buffer="1" name="moving dot" input0="0">
  <global path="movingdot.glsl"/>
</pass>
<pass buffer="2" name="downscaled dot" factor="2" input0="1">
  <global path="downsample.glsl"/>
</pass>
<pass name="draw to screen" input0="2">
  <global path="toscreen.glsl"/>
</pass>
</template>

```

This introduces 2 new buffers, one of which at **factor="2"**, meaning **screen resolution / 2**.

movingdot.glsl

```

#version 410

uniform vec2 uResolution;
uniform sampler2D uImages[1];
uniform float uBeats;

const float PI = 3.1415926535897;
const float TAU = PI+PI;

out vec4 outColor0;

void main()
{
  float dot = smoothstep(0.4, 0.3, length((gl_FragCoord.xy * 2.0 - uResolution) /
uResolution.y + vec2(sin(uBeats * TAU), cos(uBeats * TAU))));
  outColor0 = mix(texture(uImages[0], gl_FragCoord.xy / uResolution), vec4(dot),
0.5);
}

```

Don't think too much about this script... It just rotates a circle on the screen and blend in the static texture.

downsample.glsl

```

#version 420
uniform vec2 uResolution;
uniform sampler2D uImages[1];
out vec4 outColor0;
void main()
{
  outColor0 = 0.25 * (
    texelFetch(uImages[0], ivec2(gl_FragCoord.xy * 2), 0)
    + texelFetch(uImages[0], ivec2(gl_FragCoord.xy * 2) + ivec2(1, 0), 0)
    + texelFetch(uImages[0], ivec2(gl_FragCoord.xy * 2) + ivec2(0, 1), 0)
    + texelFetch(uImages[0], ivec2(gl_FragCoord.xy * 2) + ivec2(1, 1), 0)
  );
}

```

This is a very useful utility to get a weighted average of the original image.

toscreen.glsl

```

#version 410

uniform vec2 uResolution;
uniform sampler2D uImages[1];

out vec4 outColor0;

void main()
{
    outColor0 = texture(uImages[0], gl_FragCoord.xy / uResolution);
}

```

As you can see we still access `uImages[0]` even though the input is bound to another buffer.

Ping-pong buffers

To reduce the number of buffers (& VRAM) required, we can reuse buffers of the same resolution! I've added another downscaled buffer and blur the animated dot in 2 passes, reusing buffer 1 for the second pass.

Another nice feature is the ability to set uniforms from the template. These uniforms are not to be animated, but allow us to reuse e.g. a directional blur shader for multiple directions.

MyTemplate.xml:

```

<template>
  <pass buffer="0" name="lines" static="1" size="128">
    <global path="lines.glsl"/>
  </pass>
  <pass buffer="1" name="moving dot" input0="0">
    <global path="movingdot.glsl"/>
  </pass>
  <pass buffer="2" name="downscaled dot" factor="2" input0="1">
    <global path="downsample.glsl"/>
  </pass>
  <pass buffer="3" name="hblur" factor="2" input0="2">
    <global path="directionalblur.glsl">
      <uniform name="uBlurVector" value="0,4"/>
    </global>
  </pass>
  <pass buffer="2" name="vblur" factor="2" input0="3">
    <global path="directionalblur.glsl">
      <uniform name="uBlurVector" value="4,0"/>
    </global>
  </pass>
  <pass name="draw to screen" input0="2">
    <global path="toscreen.glsl"/>
  </pass>
</template>

```

Probably setting the blur vector to 4 pixels will introduce glitches, but at least it'll be blurry!

directionalblur.glsl

```

#version 410

uniform vec2 uResolution;
uniform sampler2D uImages[1];

uniform vec2 uBlurVector;

out vec4 outColor0;

void main()
{
    // 5-tap directional blur
    vec4 color = vec4(0);
    for(int i = -2; i < 3 ; ++i)
    {
        color += texelFetch(uImages[0], ivec2(gl_FragCoord.xy + uBlurVector * i), 0);
    }
    outColor0 = color * 0.2;
}

```



As you can see out-of-bounds samples with texelFetch result in black being mixed in from the borders, but then we know it works!

Multiple outputs

Probably the most important thing is the ability to return multiple outputs. This allows a single shader (e.g. a raytracer) to return lots of data, like depth, normal, world position & material properties (gbuffer).

Here I output 3 dots, from 1 shader, into separate buffers and composite them together at the end. Note how only 1 dot is blurred.

MyTemplate.xml:

```

<template>
  <pass buffer="0" name="lines" static="1" size="128">
    <global path="lines.glsl"/>
  </pass>
  <pass buffer="1" outputs="3" name="moving dot" input0="0">
    <global path="movingdot.glsl"/>
  </pass>
  <pass buffer="2" name="downscaled dot" factor="2" input0="1">

```

```

    <global path="downsample.glsl"/>
</pass>
<pass buffer="3" name="hblur" factor="2" input0="2">
    <global path="directionalblur.glsl">
        <uniform name="uBlurVector" value="0,4"/>
    </global>
</pass>
<pass buffer="2" name="vblur" factor="2" input0="3">
    <global path="directionalblur.glsl">
        <uniform name="uBlurVector" value="4,0"/>
    </global>
</pass>
<pass name="draw to screen" input0="2" input1="1.1" input2="1.2">
    <global path="toscreen.glsl"/>
</pass>
</template>

```

Note that passes using the same buffer **have** to re-specify the number of outputs. If no specific sub buffer is selected, the input always uses output 0, like 1.0 in this case.

movingdot.glsl

```

#version 410

uniform vec2 uResolution;
uniform sampler2D uImages[1];
uniform float uBeats;

const float PI = 3.1415926535897;
const float TAU = PI+PI;

out vec4 outColor0;
out vec4 outColor1;
out vec4 outColor2;

void main()
{
    float lines = texture(uImages[0], gl_FragCoord.xy / uResolution).x;

    vec2 uv = (gl_FragCoord.xy * 2.0 - uResolution) / uResolution.y;

    float dot0 = smoothstep(0.4, 0.3, length(uv
        + vec2(sin(uBeats * TAU), cos(uBeats * TAU))));
    float dot1 = smoothstep(0.4, 0.3, length(uv
        + vec2(sin(uBeats * PI), cos(uBeats * PI))));
    float dot2 = smoothstep(0.4, 0.3, length(uv
        + vec2(cos(uBeats * TAU), sin(uBeats * TAU))));

    outColor0 = vec4(mix(lines, dot0, 0.5));
    outColor1 = vec4(mix(lines, dot1, 0.5));
    outColor2 = vec4(mix(lines, dot2, 0.5));
}

```

The important part of this code is the multiple “out” variables, and their order.

toscreen.glsl

```

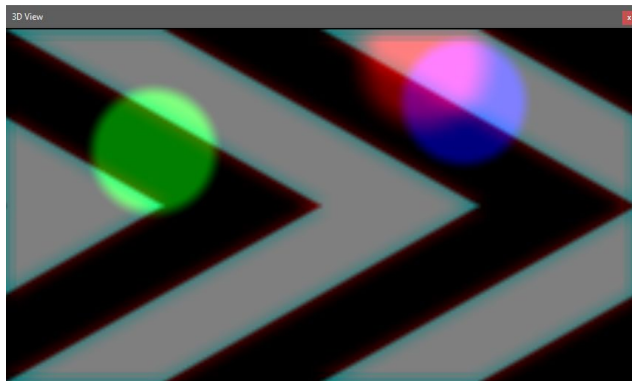
#version 410

uniform vec2 uResolution;
uniform sampler2D uImages[3];

out vec4 outColor0;

void main()
{
    outColor0 = vec4(
        texture(uImages[0], gl_FragCoord.xy / uResolution).x,
        texture(uImages[1], gl_FragCoord.xy / uResolution).y,
        texture(uImages[2], gl_FragCoord.xy / uResolution).z,
        1.0);
}

```



Animation processor

This is about procedural animation in a project. Often it is template specific, but not always. This is a very “creative” solution to get around limitations in the tool, or avoid implementing an entire reusable part in the application just for one project.

Next to the project file you can create an “animationprocessor.py” file. This file will then be evaluated (in context) before rendering a frame.

With in-context I mean it is evaluated amidst other code, python can do these things. You will find there is a “uniforms” dictionary to read and write from, these are the evaluated animation values ready to be sent to the shader.

Also there is a cameraData struct (an instance of CameraTransform in scene.py) representing the current camera state (takes into account the animation toggle).

This is the core template we use for most demos:

```

import cgmath
cgmath.prepare()
r = cgmath.Mat44.rotateY(-cameraData.rotate[1]) *
cgmath.Mat44.rotateX(cameraData.rotate[0]) *
cgmath.Mat44.rotateZ(cameraData.rotate[2])
uniforms['uV'] = r[:]
uniforms['uV'][12:15] = cameraData.translate

from math import tan
tfov = tan(uniforms.get('uFovBias', 0.5))
buf = scene.frameBuffers[scene.passes[-1].targetBufferId]
bufferWidth = buf.width()
bufferHeight = buf.height()
ar = bufferWidth / float(bufferHeight)
xfov = (tfov * ar)
uniforms['uFrustum'] = (-xfov, -tfov, 1.0, 0.0,
                        xfov, -tfov, 1.0, 0.0,
                        -xfov, tfov, 1.0, 0.0,
                        xfov, tfov, 1.0, 0.0)

```

It converts **cameraData** into a view matrix used to build rays.

It uses a **uFovBias** uniform to calculate a frustum. We used to do this inside the shader for every pixel, but with this method we can reuse these calculations if we ever want to do polygon rendering.

There is also a **deltaTime** local variable, in the event you wish to implement explicit animation, for example spring systems following an animated point.

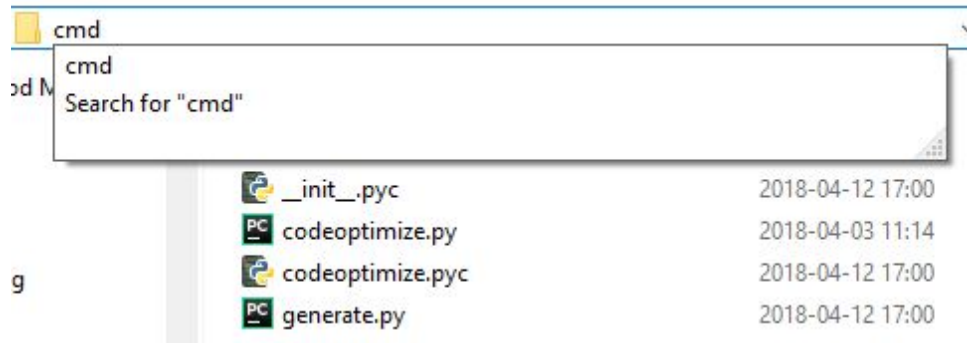
Note that the player code requires this to be reimplemented in C++ as well. There is no automated mechanism to translate this python to something useable in a 64k binary.



Creating an executable

The final step in creating a 64k! Our code is mostly autogenerated by “**build/generate.py**”. Pro-tip: open generate.py and find the call **roundb(v, 32)**, changing that 32 to a lower number will start truncating floating point data. I found half precision to be adequate for all animation data except uOrigin.

To run it, open a command prompt inside the build folder (by typing cmd in the address bar):



And run :

```
python generate.py
```

Running this script will find the last open project and write to "**Player/generated.hpp**".

Inside **Player/** you will find a visual studio project. It should build with VS2015.

Inside **Player/main.cpp** are some things you may wish to adjust:

Resolution is hard coded:

```
#ifndef DEMO_WIDTH
#define DEMO_WIDTH 1280
#endif
#ifndef DEMO_HEIGHT
#define DEMO_HEIGHT 720
#endif
```

Audio:

Is on by default, but you can disable it and jump to a certain point in the demo for debugging.

```
// #define NO_AUDIO
#ifndef NO_AUDIO
    #define BPM 124.0f
    #define START_BEAT 0.0f
    #define SPEED 1.0f
#endif
```

Loading bar:

There is a hard coded fragment shader in **const char* loader** that draws a loading bar. Feel free to change the shader behind it. The loading bar gets a **uniform vec4 u** with a [0-1] loaded state, width in pixels, height in pixels, fade (for some reason we fade it in over a few frames before we start precalc).

The last "loading" step is drawing a first frame to make sure all textures are precalced before we start the music, so expect the loader to hang for a while at that point.

Third party dependencies

The visual studio project requires the player code from 64klang2:

<https://github.com/hzdgopher/64klang/>

And add .h and .cpp files from

<https://github.com/hzdgopher/64klang/tree/master/Player/Player>

To *SqrMelon/Player/64klang2/*

In release mode the executable is compressed with kkrunchy (version a), download from here:

<http://www.farbrausch.de/~fg/kkrunchy/>

And add *kkrunchy.exe* to *SqrMelon/Player/*

Last you'll need an actual soundtrack. A musician with the 64klang2 VSTi should be able to export the patch and record note data to these 2 files:

64k2Patch.h

64k2Song.h

Which you then also add in *SqrMelon/Player/64klang2/*

*Where *SqrMelon* is the repository root

Technical reference

This is just a quick overview of what UI element lives in what python file. Code is subject to change and I only wish to provide a handle on "where to start looking" in this document.

Python's entry point is *__init__.py* in the project root, way at the bottom is a *run()* function that will start the application.

The main *App* class (also in *__init__.py*) represent the window and all UI elements are parented under it. The constructor is responsible for instantiating all sub elements and creating connections in between these elements. General best practice is to use Qt signal and slot connections to make UI components communicate without dependencies (if the connection is broken the UI component may still function).

I totally break this rule in the 3D view (*SceneView*) component, it knows about the *ShotManager*, *Camera* and *Overlay* components.

SceneList

SceneList in *scenelist.py*, it does not do any hotloading and requires an external *projectOpened* call to initialize all items shown. Scene creation and deletion also update the data model of the scene list to stay in sync, but file creation & deletion outside the tool is ignored.

ShotManager

ShotManager in *shots.py*, it handles deserialization of scene.xml files and puts the result in a table model. The first column of items contains references to actual shots.py/Shot instances which is where animation data evaluation starts.

TimeSlider

TimeSlider in *timeslider.py*, this handles the time range and the same file contains a *Timer()* class that acts as a central hub between all components that wish to set time or listen to time changes,

Camera

Camera in *camerawidget.py*, is both data and UI in one and has functions to handle mouse and keyboard input that gets appropriately integrated into the internal data. These functions are called by the 3D view.

Overlays

Overlays in *overlays/__init__.py*, analyzes the directory in the constructor and lists all png files found. The 3D view will query the overlay image and color (if any) when necessary.

3D View

Python Log

PyDebugLog in *__init__.py*, a simple text input that intercepts *sys.stdout* and *sys.stderr*.

CurveEditor

CurveEditor in *animationgraph/curveview.py*. This in turn creates a lot of child widgets, including the actual curve viewport class *CurveView*, in the same file.

Profiler

Profiler in *profileui.py*, this manages draw call profiling and debugging. Not to be confused with *profileutil.py*, which is a utility to generate python profiles and feed them to an app called *qcachegrind* (<https://sourceforge.net/projects/qcachegrindwin/>).

Known issues

Application crashes if it finds GLSL errors on startup (after showing the GLSL compiler errors in a dialog).

Specifying uniform values in the template XML means that those uniforms can no longer be animated properly. This is ok design but unclear and inconsistent between tool and player code.

Undocumented features

Just mentioning some stuff for people digging through the code.

3d texture support (has not been used since early 2017, possibly broken). Set `is3d="true"` on a static render pass in the template XML. It'll render a `size*size` by `size` 2D texture and then cast to 3D later. **If** this still works, you can uncomment a `#define` in `main.cpp` to add support in the player code.

Vertex shader support & draw call overrides. A remnant of never pulled through polygon support. Rename all `.glsl` files to `.frag` files, add `.vert` files in the template and you'll get a program with both vertex and fragment stages. Then insert a `drawcommand="glRecti(-1,-1,1,1);"` attribute on a render pass in the template XML to override what geometry is drawn (shown is the default). **This is not supported in the C++ player code in any way.**

