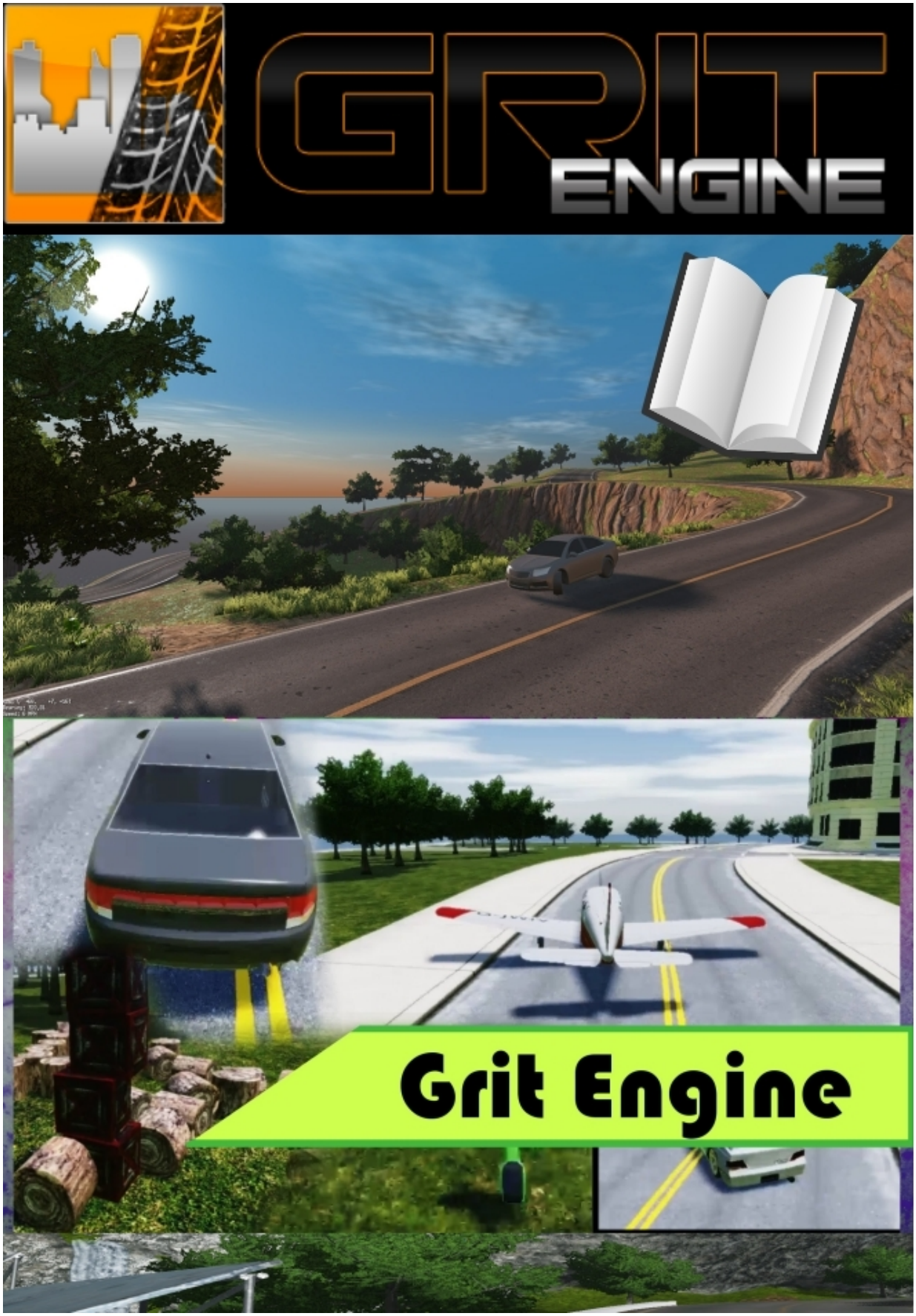


Table of Contents

Table of Contents

1. Fundamentals

1.1. Introduction

Grit Screenshot

The 'playground' map.
(TODO: This book is incomplete, but is posted online in the hope that its existing content be useful.)

The Grit Game Engine allows you to create interactive 3D environments such as computer games. Its fundamental architecture is based around the Lua scripting language. Games are created by writing scripts that have complete control all aspects of the engine, i.e. graphics, physics, game logic, audio, etc. This is done at a very high level, with an easy-to-use but powerful programming model. This book explains how to do this, by giving an introduction to the various APIs involved. The in-game Lua console allows experimentation, debugging, and development of these scripts while the game is running.

Grit is very general-purpose. This is achieved by having independent Lua APIs for the various subsystems (graphics, physics, audio, input, etc). The low level performance critical parts of the engine, such as the rendering pipeline and the collision detection, are written in C++. However, the Lua APIs are designed to be as expressive and safe as was possible without compromising on performance. Most of the APIs are object-oriented in style.

Some of these APIs allow one to create bodies, which represent some tangible thing, that can be seen, heard, or otherwise experienced by the player in some way. Lua code can modify fields of these objects in the usual way, controlling the way that they manifest, e.g. changing their colour (for graphical bodies) or their volume (for audio bodies). There are many such bodies and they each have many modifiable fields, according to their function. There are also other APIs that are less tangible like user input and resource management. Finally there is a category of Lua code that is very declarative and is approachable even for those with no programming experience. This is used for defining resources, like graphical materials.

Such APIs are a toolbox for achieving what you want, but there is another category of Lua code that allows one to create objects with their own behaviour. This is done by defining classes, and instantiating them as objects. Such objects have user-defined methods and fields like an object-oriented language. One can define game classes that instantiate into game objects. Such objects act as glue to bind together graphics, physics, audio, etc. bodies, as well as implement game logic. One can write a car game object class that handles all the simulation aspects of a car, as well as graphical details (e.g. exhaust smoke) and gameplay details (e.g. hit points), engine noise, and so on. The same technique is also used to give custom behaviour to particles, and for the GUI.

Finally, as the whole engine is open source, you can also modify the C++ code and either add functionality that way. However, this is advised if it is not possible to proceed by writing Lua scripts. For example, if you need to write a very optimised algorithm, using multicore, SIMD, and cache awareness, you would need to write it in C++. Even so, you would probably add a Lua binding for your new code, because developing high level code in Lua, using the in-game console is far easier than rebuilding the whole engine and debugging it in a C++ debugger. This book concentrates on the Lua APIs, there are some doxygen docs for the C++ code.

1.2. Licensing

Grit is open source and is released under a liberal license. It can be used for both commercial and non-commercial projects. However this does not apply to some of the example assets. It is not expected that third parties will be re-using and reselling these assets, as they exist only to demonstrate the engine. Furthermore, unlike source code, it is hard to find good quality assets under permissive licenses.

The copyright and license for the code and assets within the project package breaks down as follows:

1.2.1. Core Works

Grit core works are sufficient to build a game, if you provide your own assets. This covers all code released as part of Grit. The works are released under the MIT license:

- The C++ code for the engine itself.
- Python and Maxscript code used in exporters.
- All Luaimg code.
- All assets and Lua code in system/.
- All Lua code in common/ and vehicles/.

```
Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation
files (the "Software"), to deal in the Software without
restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or
sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.
```

1.2.2. Example Content

Some assets have copyright owned by project contributors, and are released under liberal licenses such as MIT or CC-BY, just like the source code. But other assets have less liberal licenses such as CC- SA, CC-NC or GPL. Furthermore, many assets are taken from stock image sites and thus copyright is still owned by those sites. In these latter cases, care must be taken to use the assets in accordance with the appropriate licenses, as dictated by their copyright owners. Each directory has a copyright.txt file that lists the legal status of each asset in that directory.

1.2.3. Dependencies

Grit is linked to a number of other libraries that have different licenses that you must also be aware of. See the documentation that comes with these libraries for more information.

- OGRE3D, MIT license, statically linked
- Bullet, BSD license, statically linked
- Lua, BSD license, statically linked
- Squish, BSD license, statically linked
- OpenAL, OS-dependent, dynamically linked

2. Game Objects

Objects and classes are fundamental to Grit. They are a convenient way to structure your game. Objects can be anything from a chunk of terrain, to a building, to a soda can. If you want to make a large world they are the only way to access Grit's streaming subsystem to efficiently determine the set of nearby objects, and background load incoming disk resources.

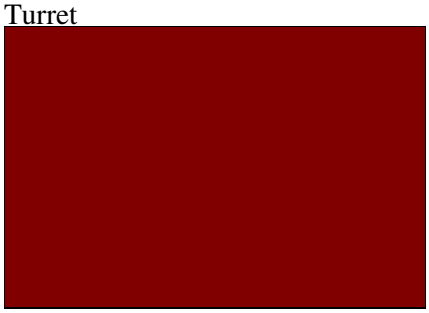
Objects can either be activated or not. Activated objects are near the camera and consume more machine capabilities (RAM, CPU, etc). In a streaming game, the majority of objects are unactivated, allowing the world to extend well beyond the horizon. All objects have at least a spawn position in 3D space and a rendering distance. Grit will use this information to activate objects as the player moves within their individual ranges.

Activated objects typically have a graphical and/or physical representation, or some active behaviour. Unactivated objects typically lie dormant, holding a tiny amount of state. There are technical limitations on what kind of data can be stored in an object when it is unactivated, for example it is not allowed to have a physical / graphical representation. This is necessary to allow scaling to large worlds.

Objects are created by instantiating a class at a particular location. The class describes the general nature of its objects. It holds the code that defines the object's behaviour, and default values for its attributes. For example, the rendering distance is typically the same for all objects of a given class, so the object inherits that attribute from its class (although the object can override it if desired). The only attribute that must be given per object is the spawn position, which is used by the streaming subsystem. All other attributes can be left to class defaults or overridden as desired. The set of available attributes depends on the class being used.

For example, a sentry turret may, as it streams in, create a graphics body for its base and another graphics body for the barrel of its gun, allowing the barrel to rotate independently to the base. It may also create a physics representation for the base (so the player can't walk through it, and to allow it to receive bullet hits, etc.). It can also create an audio body for playing whirring noises as it moves. The code to create all of this state is put in a callback called activate. There is another callback, deactivate that is called as the object streams out, and typically destroys these bodies. Thus all of this extra state exists only while the turret is close enough to the player to be visible.

Turret



Turret with a rendering distance of 100m.

As a trivial example, a ball of energy that floats up and down could be implemented with an object. The game designer writes a class with an initialisation callback that creates a graphics body in the graphics world, and stores a pointer to it in a field of the new object. The class also has a destruction callback that can be used to clean up the graphics body when the object is removed from the scene. Finally, there is a per-step callback to move the ball around, presumably using a sin function or similar. If a light source is desired, then a light body would be created in the graphics world as well, and e.g. its intensity can be modified in the step callback. Similarly, other graphics bodies, particles, and physical behaviour can be added if desired. Audio bodies can be added in the audio world. All of these "bodies", which manifest in independent subsystems, are created and controlled by the object in question.

3. Resources

Resources are things like `/foo/bar/MyThing`, that are named using a path from the base game directory. They may or may not actually exist as files on disk. Typically resources either represent assets defined in Lua files, such as classes, objects, materials, etc. (covered in the various other chapters of this book). Or, they represent assets on disk, such as sound files, textures, meshes, etc. The latter, disk resources, are the subject of this chapter.

3.1. Disk Resource Management

Disk resources can either be loaded or unloaded, which means whether or not they are occupying RAM (this can be CPU RAM or GPU RAM). Loading usually takes a noticeable amount of time, so it's a bad idea to do it during gameplay. When a body is created that depends on a resource, e.g. a GfxBody depending on mesh and texture resources, those resources will be automatically loaded (if needed). However, loading the resource immediately before creating the body will introduce a noticeable delay.

In a streaming game, the objects declare their required resources (usually in the class) and these resources are loaded in a background thread. The object is not activated until the resources are available. In this situation you rarely have to worry about loading / unloading resources. However in a non-streaming game it can be simpler to load everything before gameplay begins.

There are a number of API calls that can be used to manually control the loading and unloading of resources. In the following calls, `name` is an absolute path. To give a relative path, use backticks (§8.1.1).

```
disk_resource_load(name)  -- Note: Raises an error if it's already loaded
disk_resource_ensure_loaded(name)  -- Load, or no-op if already loaded
disk_resource_loaded(name)  -- Test if loaded
disk_resource_reload(name)  -- Useful during development to see new assets
```

Using the above calls, it is possible to load resources way ahead of time. This is useful in a game that is not large enough to need streaming, as you can load everything before the actual gameplay begins, or switch out resources between levels. However there is a catch; Grit manages the available memory and GPU memory. If, upon load, there is not enough CPU or GPU RAM, Grit will unload the least recently used resource(s) to make space for the one being loaded. A resource is used if it has at least one user:

```
disk_resource_users(name)  -- How many users does a resource have?
```

If you use a resource in some explicit way (e.g. create a GfxBody that uses a mesh), then Grit will automatically ensure it is not unloaded by increasing the user counter for that resource. However, if you want to prevent unloading during a time when the resource is not actually used, you need to tell Grit that you're really still using it. To tell Grit that you're using a resource, you create and keep a hold on it:

```
myhold = disk_resource_hold_make(name)
-- Resource will not be unloaded by Grit but may not be loaded yet
disk_resource_ensure_loaded(myhold.name)  -- Load only if unloaded
myhold:destroy()
-- Now Grit may unload it if there is a RAM shortage
```

```
-- Or you can force the unload now:
disk_resource_unload(name)  -- Works only if users == 0
```

Note that creating a hold on an object does not load it, but it will stop it being unloaded by the system until the hold is released. Loading a resource when you don't have a hold on it can be dangerous because Grit may unload it again, causing a stall when you next use it. The idiom is therefore always to create the hold, then load the resource, keeping the hold until you don't need to use the resource anymore.

3.2. Disk Resource System

The following calls allow you to query the state of the system itself:

```
disk_resource_num()  // Count all disk resources.
disk_resource_num_loaded()  // Count all loaded disk resources.
disk_resource_all()  // List all disk resources.
disk_resource_all_loaded()  // List all loaded disk resources.
host_ram_available()  // CPU RAM cap (configured by user).
host_ram_used()
gfx_gpu_ram_available()  // GPU RAM cap (configured by user.
gfx_gpu_ram_used()
```

The following flag helps debug stalls during rendering. It enables a warning if resources are loaded in the foreground thread. In non- streaming games, you can turn this on after loading your level to ensure that you loaded everything ahead of time. In between levels, you can disable it, do your unloading/loading work and then re-enable it. In streaming games, you can leave it on while the game is streaming to ensure you haven't forgotten to declare resources used by your objects.

```
option("FOREGROUND_WARNINGS", true)  -- Enable warnings if resources are loaded too late
```

4. Graphics

The graphics subsystem is the largest, and its API gives control over everything you see on the screen. This includes objects, particles, and lights in the graphics world. It also includes the heads up display (HUD) and fonts.

4.1. Graphics Bodies

4.1.1. Materials

4.2. Realtime Shadow Techniques Used By Grit

4.2.1. Depth Shadow Mapping

Grit has fully dynamic shadows that are calculated in real time on the GPU. The basic technique is called depth shadow mapping. It involves rendering the scene from the light (the sun or the moon) into a texture, called a depth shadow map. The shadow map is updated every frame, because objects move and so does the light. The colour of the scene is not computed, as we are only interested in the distance to all the surfaces that the light can 'see' (these are the occluders).

When the scene is rendered from the player's point of view, this shadow map is used as a reference to help decide if a given pixel is the closest one to the light (in which case it is not in shadow) or whether there is something else that is closer (in which case it is rendered darker because it is in shadow).

4.2.2. Perspective Transform

The depth shadow map has a limited resolution, so in order to increase the apparent fidelity (and avoid blocky artefacts) there is a perspective transform applied in order to concentrate as many as possible of the shadow map's texels close to the player. There are many techniques but the one used in Grit is called LiSPSM (LIght Space Perspective Shadow Mapping). The worst case is when the sun is directly behind you, in which case no perspective transform can be applied, and the shadow is very low detail and noisy. However, if you look 90 degrees to the sun, the shadows will be a lot crisper due to the use of LiSPSM. Note that increasing the resolution of the shadow map texture will also make the shadows crisper, but will cost memory and performance.

The perspective transform changes every frame depending on the light direction and the chase cam's direction. Sometimes the changes can be quite severe. This causes an unavoidable 'crawling' effect in the shadows.

4.2.3. Covering Larger Distances

There are in fact 3 shadow maps used. One for the area closest to the player, one to cover the area further away, and the 3rd one for the furthest reach of the shadow (200 metres). They are all the same size

textures, but the one nearest to the camera covers a much smaller area and thus the shadows are better defined. Another way of looking at this is that it allows shadows to appear much further from the player, without compromising the quality of shadows near the player. The exact technique used in Grit is called PSSM (Parallel Split Shadow Mapping). Sometimes you can see the transition from one shadow map to the next, as a sudden decrease in shadow quality.

4.2.4. Soft Shadows

If each screen pixel was merely tested for being in shadow or not, the shadows would be very hard-edged because of the sudden transition from 'in shadow' to 'not in shadow'. To avoid this, we soften the shadows using a technique called PCF (Percentage Closer Filtering) . This boils down to testing the shadow map several times per screen pixel, and taking the average. The appearance is that several faint shadows have been overlaid in slightly different positions, to produce a blurred effect. It can get very slow but there is hardware support that we are currently not using that can help, see issue 125.

4.3. Shadow Artefacts

There are certain things that can go wrong with dynamic shadow implementations like the ones used in Grit. There are some things to avoid when modelling objects, in order to avoid problems.

4.3.1. Holes in shadows

Since the shadows are calculated by rendering the scene from the sun (or moon) you have to make sure that your geometry, when viewed from this direction, appears to be opaque. This means cliffs must have polygons around the back facing the sun, in order to the sun shining through them to the front. A more expensive alternative is to turn on the rendering of backfaces in the material.

If your map is an island that drops below sealevel in all directions, you don't have to worry about this. But if your map is surrounded by some sort of "wall", then you do.

4.3.2. Shadow Texture Stretch

Using SHADOWYNESS

Shadow texture stretch occurs where polygons do not face the light.
Since the shadow texture is projected onto the scene from the light, surfaces that are perpendicular to the light (e.g. flat ground at sunset) will experience very bad texture stretch. This causes aliasing artefacts. Because of the LiSPSM perspective transformation, the artefacts have a very nasty sawtooth appearance,

instead of the square pixelation that usually occurs with aliasing artefacts.

To visualise the aliasing, we can use the following, which renders just the projection of the shadow map onto the scene, with equal intensitity for all triangles:

```
debug_cfg.falseColour = "SHADOWYNESS"
```

A small fern on the edge of a cliff is projecting a shadow downhill away from the edge of the cliff. The shadow is very elongated because of the low sun. One can see the sawtooth artefacts in the stretched part of the shadow. When animated, the moving sun causes these sawtooth artefacts to crawl in a very distracting way.

Using SHADOW_MASK

Shadow texture stretch is usually hidden by the lighting equation.
Luckily these areas should receive very much light due to the diffuse lighting equation. E.g. if the light is incident at 15 degrees then the amount of lighting would only be 25% (i.e. sin(15)) of the amount of light that it would receive at 90 degrees. This means the shadow is much less distinct in these areas. The following falseColour value shows the actual shadow, i.e. incorporating the diffuse lighting component as well as the shadow map:

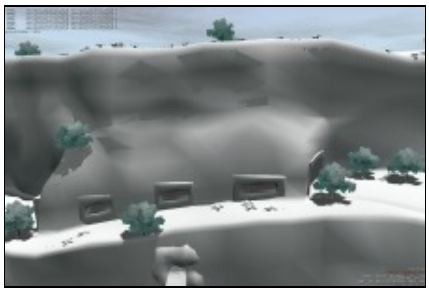
```
debug_cfg.falseColour = "SHADOW_MASK"
```

In the next section, we can see how this effect can be disrupted by certain kinds of assets.

4.3.3. Normal Bending

If your mesh has sharp edges between polys (an angle of more than 20 degrees for example) and is smooth shaded, then for some pixels, the normals interpolated across that mesh will be considerably different to the 'true' normal of that face (i.e. the normal you would calculate using the positions of the 3 vertexes). For example, if you model a cube and use smooth shading then the normals of each face will be orthogonal, but the normals will be smoothly interpolated around the cube causing a huge amount of normal bending at the edges and corners.

Normal Bending

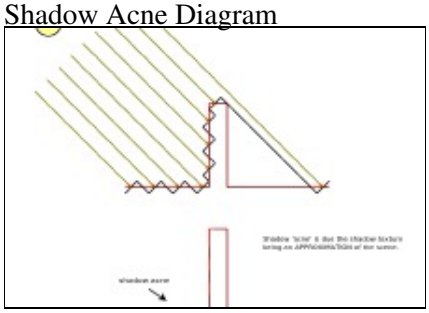


Shadow artefacts caused by normal bending.
Normal bending is usually OK, but causes a problem with shadows. This is because shadow stretch occurs in places where the true normal of the polygon is close to perpendicular to the light source, however light attenuation uses the interpolated normal, which can be pointing closer to the light than the true normal. This kind of artefact often occurs on sharp terrain like cliffs. It causes areas to be illuminated when they would not otherwise be, and therefore causes shadow artefacts to appear that would ordinarily be hidden in the darkness. If the face is in-line with the light, e.g. cliffs at noon, and there is significant normal bending, then the polygon may be almost fully lit, even though the polygon is nearly at 90 degrees to the sun.

There used to be a material property shadowObliqueCutoff for controlling this effect, but it is no-longer implemented since the switch to deferred shading. The technique was to attenuate shadows more aggressively on surfaces rendered with the material in question. However doing this on a per- material basis causes areas of the mesh that do not have normal bending to be subject to the same attenuation of shadows. The preferred solution is to calculate the amount the amount required at each vertex and store that in the vertex as an attribute. This can be fully automated in the asset generation pipeline. However it is not yet implemented. Please ignore the artifacts for now.

4.3.4. Shadow Acne

Shadow Acne Diagram

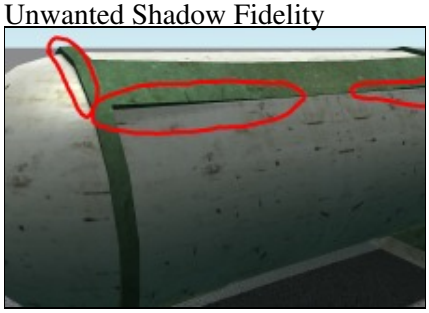


An illustration of shadow acne.
Imprecision in the shadow map, which records the distance of each occluder from the light, causes the shadow to fluctuate, causing unpleasant high frequency transitions from 'in shadow' to 'not in shadow' on every surface that faces the light. The engine will avoid shadow acne by adding a certain amount of bias to the depth of hte caster during the shadow casting phase. Thus, the shadow is pushed away from the light by enough in order to avoid the noise being an issue. The following image illustrates the problem and how the depth bias solves it, the screen shot is of a natural gas tank, but the diagram below is for a wall on flat ground.

The engine tries to use the minimal amount of bias to avoid shadow acne, by using the normal of the casting surface and a small constant offset on everything. Thus, you don't need to worry about this as a modeller, unless your surfaces are so thin that even this small amount of bias is too much.

4.3.5. Additional Bias

Unwanted Shadow Fidelity



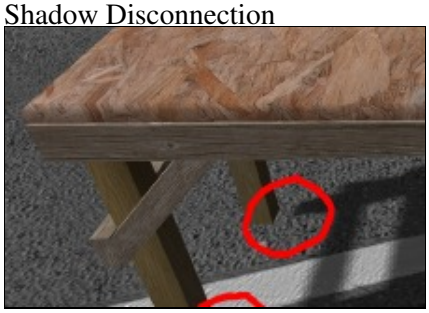
Unwanted self-shadowing.

You can add additional bias yourself, in the material, in order to get rid of other artefacts. For example here there are unwanted shadows on the tank. There is simply not enough fidelity in the dynamic shadows to properly render shadows for such detailed geometry. We would rather there were no shadows at all.

One way to avoid this is to avoid these kind of nooks and crannies in the geometry of the object. However since these contribute greatly to the appearance of objects, this may be unacceptable. Another solution is to add another 0.1m to the depth bias during shadow casting (on top of the small amount calculated by the engine), in order to push the shadow far enough away from the object to hide shadows on the high detail parts of the mesh.

4.3.6. Shadow Disconnection

Shadow Disconnection



Too much bias causes the shadow to disconnect from the base of the object.

Too much bias can cause a problem in itself though. If the bias is increased enough, the shadow will move so far from the object that there will be a 'gap' where the object meets the ground. This gives the unwelcome appearance that the object is 'floating' above the ground, as seen with this table. If you want a lot of bias, you may have to thicken the geometry of your model.

The bias automatically used by the engine is carefully chosen to be as small as it can be. However as a modeller you must also make sure your additional bias is not too large as well.

4.4. Sky Bodies

The sky is rendered in a special way. The scene is rendered first, including lighting (but not post-processing effects like bloom), and then any areas that remain undrawn are filled with the sky. The sky has only emissive lighting, as it is too far away to be affected by lights in the scene. It also does not move, it is always centered at the camera.

The sky is composed of a number of layers, composed via alpha blending in HDR buffers. Conceptually, each layer is a Sky Body whose polygons either completely or partially surround the camera (which is at 0,0,0). Thus, the body does not have a position like the regular graphics body (§4.1), although it can be rotated.

Different layers allow the controlling of different effects, e.g. you can have a base stars layer, then a moon layer (which you rotate to control the position of the moon), followed by a sky layer (with some sort of atmospheric colour model), finally followed by clouds. Each of these layers can be enabled/disabled and oriented seprately.

Each sky body uses a mesh that is exported from a modeller in the usual way. It is typically a cube or sphere, or in the case of sun / moon etc, a small billboard. One can also imagine strips or other shapes being used in certain cases, like meteorite belts or jet trails. The polygons all point inwards, towards vec(0,0,0). The distance of the polygons from 0,0,0 can be arbitrary, since the depth buffer is not used to control the layering of sky bodies, rather they are layered explicitly. Typically we use about 1 metre because that is easy to manipulate in modelling software.

Sky body materials refer to sky materials instead of regular materials (§4.1.1). Sky materials are defined in a special way, and are not interchangeable with regular materials, although they can share textures. Here is an example:

```
sky_material `starfield` {
    emissiveMap = `starfield.dds`;
}
```

There is currently implemented a prototype for custom sky shaders, that interact with the sky materials. This is going to be changed in future to make it easier to use, so we shall not document it here. Take a look at /system/sky.lua if you are curious to see the current state. If you need advanced skies now, contact us via IRC or the forums for assistance.

The sky body itself is created as shown below:

```
sky_ent = gfx_sky_body_make(`MySkyMesh.mesh`, 255)
sky_ent.enabled = true/false -- disable drawing of this layer
sky_ent.zOrder = 100 -- change z order
sky_ent.orientation = quat(1,0,0,0)
```

The 255 is the z order, an integer between 0 and 255 inclusive. The value of 255 places it behind everything else. Use lower numbers to add more layers on top. If two sky bodies have the same z order, the ordering is undefined. So don't do that unless you know the polys will not overlap.

4.5. Lights

Lights provide illumination of the scene (beyond that of the sun/moon) and also provide a corona sprite. They exist at a given place in the scene and (if set up to be spotlights) can be rotated to light in a given direction. They are implemented using deferred shading, which means their performance cost is proportional to the number of screen pixels lit by them. This allows lots of small lights to be rendered efficiently.

The lights are nodes in the scene graph, so can be attached to other nodes and have the same fields. Like graphics bodies, they can be individually enabled/disabled and faded.

Light Diagram

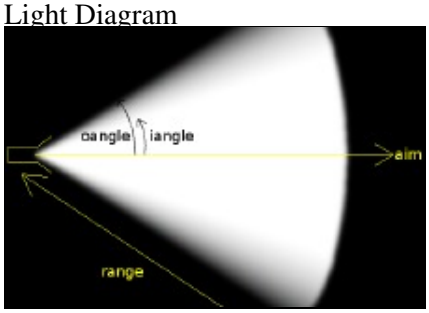


Illustration of various light fields.

```
l = gfx_light_make()
l.localPosition = vec(1, 2, 3)
l.orientation = quat(1, 0, 0, 0)
l.diffuseColour = vec(2,2,2) -- note use of HDR
l.specularColour = vec(2,2,2)
l.range = 10 -- light extends for 10m
l.innerAngle = 50  -- see diagram
l.outerAngle = 60
```

The outerAngle must be >= the innerAngle, as shown in the diagram. If the innerAngle is 180, then the light is not directional, so the orientation does not matter. Otherwise it shines in the +Y direction unless oriented differently.

4.5.1. Coronas

If desired, Grit can draw a corona to simulate the blinding effect of the light itself. The corona's size (diameter) must be specified, as it defaults to 0 which means no corona. The corona also has its own colour, which you must set otherwise it defaults to white.

```
l.coronaSize = 0.3 -- sphere diameter in metres
l.coronaLocalPosition = vec(0, 0, 0)
l.coronaColour = vec(1,1,1) -- white
```

4.6. Particles

Particles are a special kind of graphical effect. Conceptually they are cubes in the game world, i.e., they have a 3D position and size, but their orientation is constrained to face the direction of the camera (and optionally to be rotated around the vector to the camera). Although they are conceptually 3D, they are rendered with textured 2D quads that face the camera. The shader uses the screen depth to soften their intersection with other geometry in the scene.

Particles are typically used to render certain types of volumetric gas-like effects like smoke, clouds, kicked up dirt, flames, and explosions. Each particle is a fixed, roughly spherical shape, but with a cloud of particles, the overall shape can be roughly approximated.

Although the term "particles" implies that there should be a large number of small particles, typically this is not the case. Particles are reasonably heavy to render and you don't want to have too many behind each other in the case of alpha particles because that causes a lot of overdraw.

Use just enough particles to represent your 3D shape. The key to particles is to use a great looking (preferably animated) texture, and then write enough behaviour into that particle class, that a cloud of particles together resemble what you want.

There are a number of particles defined in /common, which you can spawn in your own code. Typically a particle is created at a given location, possibly with customised values for some other parameters , and then from that point on it is in charge of its own destiny, including destroying itself. Particle behaviour can be arbitrarily complex, performing physics tests or interacting with objects in the scene. They can even spawn more particles.

(TODO: The particles API needs some cleanup before it is worth documenting.)

To create your own particles, define a Particle Class in Lua scripting, which provides the references textures, default fields, and provides a stepCallback to implement the behaviour of the particle.

Particle texture coordinates are used to implement animation. When defining a particle, a list of coordinates are given, one for each frame.

Particles have a number of other attributes that define their appearance: colour, alpha, position, angle, and size.

4.7. The Heads Up Display (HUD)

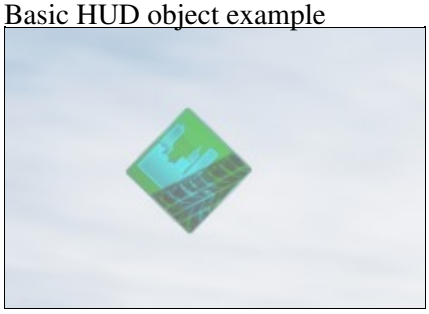
The heads up display (HUD) is a 2D framework for providing a graphical user interface to the player. Like everything else, it can be extensively customised via Lua. The most basic API allows positioning and rotating text / textured rectangles in a scenegraph-like hierarchy on the screen. On top of this, various widgets have been implemented in Lua, via HUD Class definitions. This, you are also free to make your own. The HUD subsystem is designed to handle things like health/ammo displays, mini-maps, menu

screens, and general purpose mouse-driven graphical user interfaces.

4.7.1. Basics

Let us first instantiate an existing class. The Rect HUD class is the simplest class it is possible to write, as it has no user-defined behaviours. Its one-line definition is in the common/hud directory, but you can also define your own. Instantiate it as follows:

Basic HUD object example



Use of /common/hud/Rect as shown in the Lua snippet.

```
obj = gfx_hud_object_add(`/common/hud/Rect`)
obj.position = vec(100, 100)  -- centre of object, from screen bottom left
obj.texture = `/common/hud/textures/Icon064.jpg`
obj.size = vec(50, 50)
obj.colour = vec(0, 1, 1)  -- cyan (masks texture)
obj.zOrder = 2
obj.alpha = 0.5
obj.orientation = 44  -- degrees (clockwise)
...
obj:destroy()
-- Can also initialise members with a table param:
obj = gfx_hud_object_add(`Rect`, {size=vec(50,50), ...})
```

The HUD object has a number of fields that define how it is rendered. Some examples of these are above, and they are comprehensively documented (TODO: here). If not given, they fall back to defaults first in the HUD class and then the system defaults.

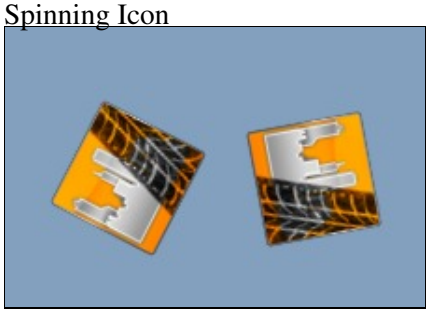
The size defaults to the size of the texture, if one is given. The zOrder field is an integer between 0 and 7, which defaults to 3. Higher values are drawn on top of lower values. The HUD object is garbage collected when it is abandoned, and can be destroyed earlier with :destroy().

Rect can be useful for simple things but usually we will define our own HUD class, which has fields and methods much like a game class. The HUD class can be instantiated into as many HUD objects as we want. The following HUD class implements a spinning icon;

```
hud_class `SpinningIcon` {
    texture = `/common/hud/textures/Icon064.jpg`;
    speed = 20;
    init = function (self)
        self.needsFrameCallbacks = true
        self.angle = 0
```

```
    end;
    destroy = function (self)
        -- nothing to do
    end;
    frameCallback = function(self, elapsed)
        self.angle = (self.angle + elapsed*self.speed) % 360
        self.orientation = self.angle
    end;
}
```

Spinning Icon



The icon on the right (faster_obj) spins faster than the one on the left (obj).
The 3 defined methods are all callbacks, and there are other possible callbacks as well. You can also define your own methods with your own names, to help structure your code or to provide a public API to the object. The init callback sets the angle field of the object and requests frame callbacks. If you enable a callback, you had better have a method of the right name to receive it. If a callback raises an error, the system will usually disable the callback by setting the needsX field to false again.

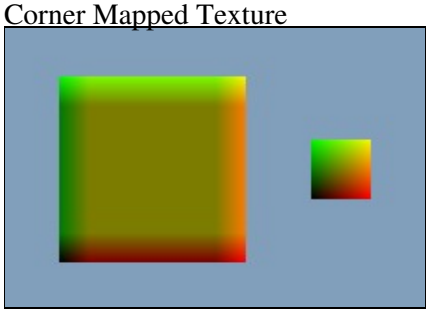
Note how giving the speed in the class allows it, optionally, to be overridden on a per-object basis. The following code shows how we can override the speed when we instantiate the object:

```
obj = gfx_hud_object_add(`SpinningIcon`, {position=vec(200,100)})
faster_obj = gfx_hud_object_add(`SpinningIcon`, {position=vec(300,100), speed=30})
```

4.7.2. Corner Mapped Textures

By default, when a texture is set on a HUD object, it is filtered (stretched or shrunk) to fit the rectangular size of the object. This can be useful when the size of the object can vary at run time, or if a single texture is to be used for a variety of differently sized HUD objects. However, the filtering artefacts can look ugly for a lot of textures, especially when the texture is magnified a lot.

Corner Mapped Texture



The texture itself is shown to the right. To the left is shown the HUD object mapping this texture with corner mapping.
For certain kinds of textures, there is a technique that avoids the artefacts while allowing arbitrary scaling of the HUD object. This involves mapping the four quarters of the texture 1:1 into the corners of the HUD object and stretching a single row and column of texels across the middle of the HUD object.

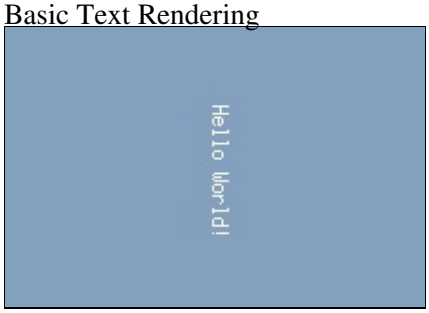
This works well in a variety of situations where the centre of the HUD object is a solid colour, but detail is desired around the edge and at the corners. For example, simple rounded corner rectangles, borders, bevels, etc. Some example textures are provided in /common/hud/CornerTextures. See the button (§4.7.6.1) class below for a concrete example.

```
o = gfx_hud_object_add(`/common/hud/Rect`, {texture=`MyTexture.jpg`})
o.size = vec(100, 100) -- larger than the texture
o.cornered = true -- enable corner mapping
```

4.7.3. Text

HUD text bodies allow the rendering of text according to a particular font (§4.8). It supports top/bottom colour, ANSI terminal colour codes, hard tabs, new lines, wrapping according to a pixel width, and scrolling.

Basic Text Rendering



A HUD text body using the simple misc.fixed font.
The text bodies behave like HUD objects, but they are not extensible. They do not have customisable fields and methods, and they cannot receive callbacks. They do, however, have all the basic fields of HUD objects.

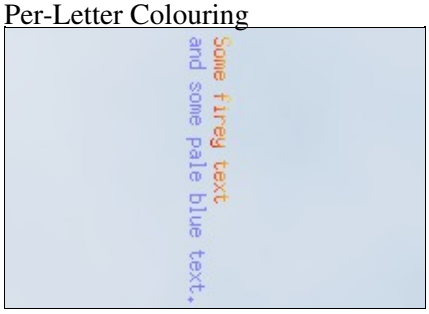
```
t = gfx_hud_text_add(`/common/fonts/misc.fixed`)
t.text = "Hello World!"
t.position = vec(300, 300)
t.orientation = 90
t.zOrder = 7
...
t:destroy()
```

The size field of text is immutable. Upon setting the text, it gives you the number of pixels occupied by this particular text in this font. The code takes account of tabs, new lines, and ANSI terminal colour codes.

4.7.3.1. Text Colour

Colour and alpha are supported via two mechanisms. Firstly, and most simply, one can set the colour/alpha fields, which act as a general mask on the colour/alpha present in the texture. This is the same as regular HUD objects.

Per-Letter Colouring



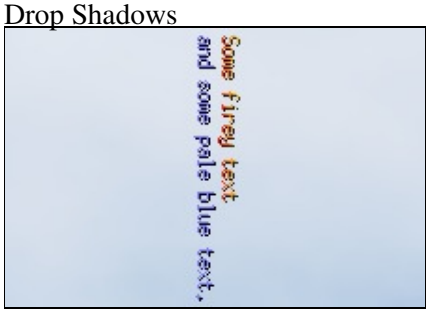
Text with per-letter colouring.

To apply colour and alpha to individual letters and words, and to use top/bottom colouring (gradients) there is a different mechanism. Instead of setting the text field, one uses several calls to append(), between which the letter colour/alpha can be changed. The clear() call will reset the state. Setting the text field is equivalent to calling clear(), setting the colours to white and the alphas to 1, and then calling append().

```
t:clear()
t.letterTopColour = vec(1, 1, 0) -- yellow
t.letterBottomColour = vec(1, 0, 0) -- red
t.letterTopAlpha = 0.7
t:append("Some firey text\n")
t.letterTopAlpha = 1
t.letterTopColour = vec(.5, .5, 1)
t.letterBottomColour = vec(.5, .5, 1)
t:append("and some pale blue text.")
```

4.7.3.2. Text Drop Shadows

Drop Shadows



A drop shadow of 1 pixel towards the bottom right of the screen.

To allow clearer reading of the text on arbitrary backgrounds, it is possible to enable a drop shadow. This works by rendering the text first coloured black, at an offset, and then rendering the actual text in a second draw on top of it. In the second case below, the texel lookup are offset by 0.5 texels, which causes bilinear filtering. This has the same effect as blurring the text with a 2x2 box filter. Note that there must be an additional pixel of space around each letter in the font texture for this to work.

```
t.shadow = vec(1, -1) -- bottom right by 1 pixel
t.shadow = vec(1.5, -1.5) -- pixel blurring
```

4.7.3.3. Wrapping Paragraphs

By default, the text body will only break the text at any new-line characters (`\n`) given in the input text. This means the width of the text body will be the length of its longest line. To automatically wrap the text at a given width in pixels, set the textWrap field, which defaults to nil.

```
t.textWrap = vec(500, 400)
```

This defines a rectangle. Firstly, text is broken to avoid it rendering wider than the given number of pixels (500 in this case). Secondly, no more lines are drawn than would fit into the given height (400 pixels in this case). The size field will always return the textWrap dimensions when it is non-nil, regardless of how much text is in the text body. Lines are broken at word boundaries unless a word is too long for the line.

It is also possible to scroll the text, i.e. to clip lines at the top of the text, as well as, or instead of, the bottom of the text. This is done by setting the scroll field, which defaults to 0.

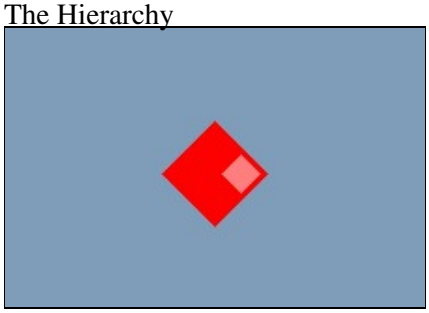
```
t.scroll = 200
```

This starts rendering at 200 pixels below the top of the given text, i.e. represents a scrolling down of 200 pixels.

4.7.4. The Hierarchy

Like the 3D scene, the HUD objects exist in a hierarchy, i.e. one can be the child of another. This is achieved by setting the parent field:

The Hierarchy



The child is drawn relative to the parent.

```
p = gfx_hud_object_add(`/common/hud/Rect`, {
    colour = vec(1,0,0);
    position = vec(400,150);
    size = vec(40,40);
})
c = gfx_hud_object_add(`/common/hud/Rect`, {
    alpha = 0.5;
    position = vec(10,10);
    size = vec(15,15);
```

```
})
c.parent = p
p.orientation = 45
```

The object p has a child object c. This relationship was established by setting its parent field, which can be changed at any time or provided in the constructor's 2nd argument. HUD objects can be children or parents, but HUD texts can only be children. A tree is thus formed, with HUD objects at the nodes, and either HUD objects or HUD text at the leaves.

The position of the child is relative to its parent. It thus displays on top of the parent, in its upper right corner. The child is always drawn on top of the parent, so setting zOrder is only necessary to disambiguate the draw order of siblings. Disabling a parent causes all children to effectively be disabled as well.

Typically, this hierarchy is used to implement more high-level GUI elements in terms of low-level ones. For example, a GUI dialog might consist of buttons and text entry widgets. Disabling the dialog would hide everything, and the dialog can be moved / rotated as a single atomic unit.

As the parent's position (or orientation) is updated, the child's derived position and orientation are recalculated. It is also possible to set inheritOrientation to false, which is useful to avoid icons rotating within a spinning mini-map, or to keep text looking crisp and clear.

Note that the child will be garbage collected if there is no link to it, so it is standard to store a link from the parent to the child in order to keep it alive. The link from child to parent is a weak link, in that it does not in itself prevent the parent from being garbage collected. Typically, however, one does not link to children without also linking to the parent.

When a HUD object is destroyed, its children are automatically destroyed with it. If you don't want this behaviour, you can unlink the children in the destroy callback by setting their parent to nil.

4.7.5. Handling Keyboard/Mouse Input

Handling keyboard/mouse input allows us to create interactive HUD objects, for example buttons, text entry, scroll bars, etc. There are two callbacks that provide this fundamental capability -- one for mouse pointer movements, and one for button events (both mouse and keyboard buttons). They are both enabled by setting needsInputCallbacks:

```
hud_class `InputDemo` {
    init = function (self)
        self.needsInputCallbacks = true
        self.inside = false
    end;
    destroy = function (self)
        -- nothing to do
    end;
    mouseMoveCallback = function(self, local, screen, inside)
        print(local, screen, inside)
        self.inside = inside
    end;
    buttonCallback = function(self, event)
```

```
        if inside then
            print(event)
        end
    end;
}
```

The mouseMoveCallback gets the cursor position local to the object (with derived orientation and position transformations reversed), the screen cursor position, and a boolean indicating whether the mouse cursor is over the HUD object or not. This takes into account other HUD objects that may be occluding this HUD object. If inside is true, then the local position should be within the rectangle defined by the HUD object's size.

The buttonCallback receives all keyboard and mouse button events whenever needsInputCallbacks is true. So typically, you should implement some code to implement mouse or click focus in order to disregard button presses when they should not be processed. Grit does not prescribe a focus strategy, but it is trivial to implement one, as shown in the above code.

The received events are strings. Each string consists of a prefix character, one of +, -, =, or :, that denotes the action. For example +a means a was pressed, -a means a was released, and =a is a repeat event, issued by the operating system because the key is being held down. For such keys, the suffix of the key is the name of the button, which is always lower case. Buttons like the arrow keys have names that begin with an upper case letter, e.g. "Left" or "Ctrl". The mouse buttons have names beginning with a lower case letter, such as "left", "middle", and "right".

Key events beginning with the ":" character are text events. These incorporate the shift key for capitalisation, internationalisation accent keys, and other keyboard modifiers that affect the way text is typed. So use these events for text input, do not try and capitalise letters yourself by watching for the shift key.

4.7.6. Existing HUD Classes

In /common/hud there are a few existing classes you can use to build GUIs quickly. /common/hud/Rect is an example, and we have already seen several example applications in this chapter (§4.7). The other classes are more sophisticated. In all cases, the code for these classes is available to be read. This is useful to find out how to instantiate them, and also to see examples of HUD code, for those writing their own HUD classes. Some key classes are documented here:

4.7.6.1. Button

The class /common/hud/Button can be used to create a clickable button with text on it. The button changes colour on mouse hover and click, and can also be greyed out. It can be instantiated as follows. Note the pressedCallback, which the code in this class executes when it receives a button press.

Button

The button created by the Lua snippet left.

```
b = gfx_hud_object_add(`/common/hud/Button`, {
    caption = "Click Me!";
    pressedCallback = function (self)
        print("You clicked me!")
    end;
})
-- By default, size is chosen based on the caption size.
b.parent = my_dialog
b.position = vec(40, 0) -- position within the parent
b:setGreyed(true) -- turn the button off, default false
b.enabled = false -- hide the button, default true
```

It is also possible to customize some of the attributes of the button. The following rather garish button demonstrates this.

Garish Button

The colours become more interesting upon mouse interaction.

```
b = gfx_hud_object_add(`/common/hud/Button`, {
    size = vec(100,40);
    font = `/common/fonts/Impact24`;
    caption = "Click Me!";
    baseColour = vec(0, 0, 0);  -- Black
    hoverColour = vec(1, 0, 0);  -- Red
    clickColour = vec(1, 1, 0);  -- Yellow
    borderColour = vec(1, 1, 1);  -- White
    captionColour = vec(1, 1, 1);
    captionColourGreyed = vec(0.5, 0.5, 0.5);  -- Grey
})
```

4.7.6.2. Label

The /common/hud/Label class encapsulates a HUD text body. It provides a simple way of aligning the text, and has the ability to set the text to grey, which is useful when greying out a whole dialog, including text

labels.

```
label = gfx_hud_object_add(`/common/hud/Label`, {
    size = vec(100, 40);
    alignment = "LEFT";  -- Also, "CENTRE", or "RIGHT".
    value = "Enter number of things here: ";
    position = vec(30, 30);
    greyed = true;
})
label:setValue("Changed value: ")
label:setGreyed(false)
```

4.7.6.3. Border

The /common/hud/Border class wraps a child object and draws a border around it using corner mapping (§4.7.2). When instantiating the class, the child object is given and automatically has its parent set to the border object.

Border

Putting a border around another HUD object.

```
o = gfx_hud_object_add(`/common/hud/Border`, {
    position = vec(200, 200);
    padding = 8;  -- Extra space on the inside.
    colour = 0.25 * vec(1, 1, 1);  -- Dark grey
    texture = "/common/hud/CornerTextures/Border04.jpg";
    -- The child object:
    child = gfx_hud_object_add(`/common/hud/Button`, {
        caption = "Some button.";
    })
})
```

(TODO: Editbox, scale, controls)

4.7.7. Layout

By making use of the HUD hierarchy, one can position HUD objects using relative co-ordinates. When creating a dialog, this means hardcoding the positions of all the elements of that dialog, which can be tedious and hard to maintain. There are a few HUD objects that control the layout of their children, making this easier.

4.7.7.1. StackX and StackY

The /common/hud/StackX and StackY HUD classes will control the positions of their child objects and arrange them into horizontal or vertical alignment. StackY will place the children vertically, in order, starting from the top. StackX does the same but horizontally, starting from the left.

Ultimately, the size of the stack object will be the bound of all the objects inside it, allowing you to nest stacks, add borders, etc. The stack cannot be directly resized, it also does not react to children whose size change at some time after the stack was created.

Border

Three buttons laid out using StackY. Note the extra gap of 10 pixels.

```
o = gfx_hud_object_add(`/common/hud/StackY`, {
    position = vec(200, 200);
    padding = 8;  -- Default space between elements (defaults to 0)

    -- The child objects:
    { align="LEFT" };  -- "RIGHT" and "CENTRE" (default) also available.
    gfx_hud_object_add(`/common/hud/Button`, {
        caption = "Shoot Missile.";
    });
    gfx_hud_object_add(`/common/hud/Button`, {
        caption = "Shoot Laser.";
    });
    vec(0, 10);  -- It's also possible to add extra space (or subtract it).
    gfx_hud_object_add(`/common/hud/Button`, {
        caption = "Place Mine.";
    });
})
```

Each element in the list is either the next child object, a vector2 value giving some padding (which behaves like an invisible Rect of the same size), or a string that specifies a new alignment. The alignment applies to all subsequent objects, unless it is changed again.

The StackX is the same, except obviously they are placed one after each other from left to right. Also, the alignment options are "TOP", "CENTRE", and "BOTTOM", and any additional spacing is given as vec(0, n) instead of vec(0, n).

4.7.7.2. Stretcher

The /common/hud/Stretcher class will position its child object to fill the given space. It does this by changing its own position (the child is at vec(0, 0) relative), and size, and also changing the child's size to fill its own boundaries. If the child has an updateChildrenSize callback, it calls that after changing the child size.

The class is instantiated by overriding the calcRect function, which allows the user to specify how the size of the stretcher should relate to its parent's size. The callback is re-executed every time the parent size changes. The result of calcRect is passed to setRect, i.e. it is the values left, bottom, right, top. In this case, it specifies a 50 pixel padding, and the Stretcher has no parent, so it uses the screen dimensions as input.

Stretcher

Using the /common/hud/Stretcher class to resize a button to fill the screen.

```
o = gfx_hud_object_add(`/common/hud/Stretcher`, {
    calcRect = function(self, psize)
        return 50, 50, psize.x-50, psize.y-50
    end;
    child = gfx_hud_object_add(`/common/hud/Button`, {
        caption = "Some button.";
    })
})
```

4.7.7.3. Positioner

The /common/hud/Positioner controls its own position according to the size of its parent. It is typically used to place other objects relative to the four corners of the screen, and update these positions as the resolution or window size changes. There are 5 existing positioner objects you can attach a HUD object to (by setting them as parent): hud_center, hud_bottom_left, hud_bottom_right, hud_top_left, and hud_top_right.

```
o = gfx_hud_object_add(`/common/hud/Rect`, {
    size = vec(100, 100);
    parent = hud_top_left;
    position = vec(50, -50);
})
```

4.7.8. Bringing it all together

4.8. Fonts

The font subsystem allows the definition of fonts that can then be used to render text in the HUD (§4.7) and, in future, in the game world (e.g., for signs, etc). Each font has a name in the directory hierarchy, so they should be referred to using the `` quotes. There are many examples in /common/fonts.

The font texture contains all the letters for a particular size. It commonly has an alpha channel to allow the text to be rendered on a background, and is typically white, so the text colour can be controlled with a mask at render time. However you can use colours if you wish. This can be useful in some cases, for big title text and so on.

A font binds from each unicode code point to a rectangle within the texture. Not every code point need be bound. If a code point is not bound in a font, then the font subsystem tries to find another character that is. First it tries the unicode replacement character "?", then space " " then finally the letter "E", before giving up and skipping the character.

To define a font, use the gfx_font_define function. This takes the name of the font being defined, the name of the texture to use, the line spacing (font height), and then a table giving the texture co-ordinates for each codepoint. Each code point defines a rectangle of texels, by giving the x, y of the bottom left corner of the rectangle, and also the width and height. The texture co-ordinates are relative to the bottom left of the texture. The following font defines codepoints for the letters A and B, and maps the same rectangles to a and b as well. This can be a useful way of making text appear in all capitals.

```
gfx_font_define(`MyFont`, `MyFont.jpg`, 12, {
    [0x0041] = {0, 0, 10, 12},  -- A
    [0x0042] = {0, 10, 10, 12},  -- B
    [0x0061] = {0, 0, 10, 12},  -- a
    [0x0062] = {0, 10, 10, 12},  -- b
})
```

In the above case, both letters are the same size, but this need not be the case. One can define variable width fonts, which are often more attractive than monospaced fonts. If the letters are different heights, they are aligned along the top of the rectangles as the text is rendered. This means you can clip unused texels at the bottom of character, saving texture space.

5. Physics

The physics subsystem exists independently of the graphics subsystem. There can be physics bodies that have no graphical representation (i.e. invisible), and equally there can be graphics bodies that have no physical representation (e.g. particles / ghosts).

The physics subsystem exposes a physics world in which physics bodies can be created to interact with each other. It is also possible to do spatial queries, like ray casts, sweeps of various shapes and collision volume tests. Callbacks and query results from the physics world are used by Lua scripts to update graphics and perform game logic.

5.1. Physics Bodies

Physics Bodies exist in the physics world. Every game step, their position and orientation are updated based on rigid body dynamics and other physical simulation. Typically, this information is used to update graphics objects but in principle it could be used for anything.

5.2. Collision Files

6. Audio

Grit's audio support supports playing wav and OGG Vorbis files from disk. Sound files can be mono or stereo. Sounds can either be instantaneous or continuous, and in addition, they can either be ambient or positional. Therefore there are four combinations.

Instantaneous sounds are usually short, they are used for bangs and throw away noises. They cannot be stopped or paused, or have their volume or pitch changed while the sound is playing. Continuous sounds are the opposite.

Ambient sounds play in the same manner everywhere. They are typically used for environmental noise. They are usually stereo, and the channels are mapped directly to the computer's speakers.

Positional sounds have a position in the 3D world, and their volume, mapping to speakers, (and possibly other effects) is controlled automatically based on the camera location and orientation. Instantaneous positional sounds play to completion at a fixed point, typically used for impact sounds and similar events. Typically these are mono sound effects. If not, then two mono sound emitters are created on top of each other. If this is not what you want, you can use an audio body, or petition us to add extra parameters to the call.

```
volume = 1  -- multiplier
pitch = 1  -- multiplier
audio_play_ambient(`Thunderclap.wav`, volume, pitch)
pos = vec(0,0,0)
ref_dist = 3  -- Distance up to which the sound plays at max volume.
roll_off = 1  -- Rate that sound attenuates beyond ref_dist.
audio_play(`Smash.wav`, volume, pitch, pos, ref_dist, roll_off)
```

6.1. Audio Bodies

Continuous sounds are implemented with audio bodies. Playback is controlled via methods and fields of these objects. Sounds can be looped, paused / continued, and stopped / restarted. The pitch and volume can be changed at any time.

```
a = audio_body_make_ambient(`BirdSong.wav`)
a.pitch = 1
a.volume = 1
a.looping = true
a:play()
a:pause()
a:stop()
...
a:destroy()
```

Audio bodies playing positional sounds are created with a different constructor. They have additional fields for updating position, velocity, etc. If the sound file is stereo, two sound emitters are created in the game world. The left and right channel are separated along the world space X axis (left channel west, right

channel east). The separation (in metres) can be controlled. There is also an orientation parameter if you want them separated in a different direction.

```
a = audio_body_make(`CarEngine.wav`)
a.position = vec(0, 0, 0)
a.velocity = vec(0, 0, 0)
a.separation = 4
a.orientation = quat(90, vec(0,0,1))  -- Align north/south.
a.referenceDistance = 3  -- See audio_play, above.
a.rollOff = 1  -- See audio_play, above.
```

6.2. Audio Options

The following global audio options can be controlled from Lua scripts:

```
audio_option("DOPPLER_ENABLED", true)  -- Whether to use doppler shift.
audio_option("MUTE", false)  -- Mute all sounds
audio_option("MASTER_VOLUME", 1.0) -- Control all volumes.
```

7. Input

In Grit, user input, in the form of mouse movements and keyboard/mouse button presses, is delivered to the scripting environment via a chain of InputFilter objects. Each InputFilter has a set of button bindings associated with it, e.g. "C+A+s" meaning Ctrl+Alt+S. These buttons include both keyboard and mouse buttons. Lua code registers callbacks to handle these button presses. The first function is called when the button is pressed down. The second is called when it is released, and the third is for key repeat events. One can give 'true' for the 3rd argument in order to use the button down callback for repeat events.

```
my_filter = InputFilter(135, "My Bindings")
my_filter:bind("C+A+s", function() print("Hello World!") end, nil, true)
```

The intent is that different modules of behaviour within Grit have their own InputFilter objects, which are set up with callbacks that interact with that code and state. Globally, the InputFilters are chained according to a order value (135 in this case). Each button press trickles along the chain of InputFilters starting with the one with the lowest order value and then trying the others in ascending order. The first InputFilter with a binding for that button has its callback invoked, and the button press is blocked from being received by the lower InputFilters.

The intention is that the input filter for global button presses (such as to quit the game) appear earlier in the chain and cannot be used to e.g. control a car. Any button presses that get to the bottom of the chain without triggering a callback are delivered to the HUD system.

In addition to these mechanisms, it is possible to disable individual InputFilters. This causes the input subsystem's behaviour to be as if the disabled InputFilter was removed from the chain. It is an easy way to control a set of bindings that only occur during a certain state of game play. For example, one can have a set of bindings for navigating through a menu system, that is only enabled if the menu system is being displayed.

```
my_filter.enabled = false
```

It is also possible to set a modal flag on a given InputFilter. This causes the input subsystem's behaviour to be as if all later InputFilters (the ones with larger order values) were disabled. Continuing our menu system example, setting modal is useful for disabling large amounts of functionality, so that e.g. a car is not being controlled while the player is in the menu system.

```
my_filter.modal = true
```

To capture mouse movement events, an InputFilter must set the capture flag. This enables the calling of a mouse movement callback for receiving relative mouse cursors movements, and prevents all input events (mouse movement and buttons) from reaching the HUD. It also hides the mouse cursor.

```
my_filter.mouseMoveCallback = function (rel) print("Mouse: "..tostring(rel)) end
my_filter.mouseCapture = true
```

InputFilters will be garbage collected, but can also be destroyed earlier as usual:

```
my_filter:destroy()
```

For debugging purposes, it may be useful to examine the complete set of currently existing input filters. This can also give a clue as to what order value to give for your input filter. The following function returns a map from order number to the description string. The only way to get a reference to the actual input filter is to find which variable it is stored in, on the Lua side, e.g. my_filter in the above examples.

```
input_filter_map()
```

8. Miscellaneous

8.1. Lua

Lua is a general purpose scripting language. It is well-known for having a small and fast implementation but a wide set of features including co-operative threading (co-routines), closures, "self"-style object orientation, and so on. Lua is more expressive and faster than all of the mainstream scripting languages and is used in many computer games.

To learn Lua quickly, read chapter 2 of its manual.

Lua is used in Grit with some minor modifications that make Lua code written for Grit incompatible with a regular Lua interpreter and regular Lua programs potentially incompatible with Grit. This is done for performance and usability.

It should be possible to make regular Lua code work in Grit with minor (probably zero) modifications and the same performance. It is very unlikely that Grit Lua code will work outside of the customized Grit Lua VM without running much more slowly.

This section details the Lua modifications. You can skim it if you are not already very familiar with Lua, and if you're not intending to port Lua code from external sources to run in Grit.

8.1.1. Relative Path Literals

It is convenient for Lua files to refer to resources via relative paths. This makes it possible to define packages of scripts and resources, that can be downloaded and unpacked anywhere in the directory hierarchy without having to edit the Lua files to correct absolute paths.

However, when using strings to represent relative paths, the strings can be passed by value through function calls and variable assignments and end up being used in a different Lua file, where the relative path no-longer resolved correctly.

In order for the relative paths to be resolved in the file where they are defined, Lua has been extended to add another kind of string literal:

```
print(`foo`)
print(`/dir/foo`)
print(`../foo`)
print(`.`)
```

This string is assumed to be a path. If the path begins with a '/' then it behaves the same as ordinary strings. However other paths are assumed to be relative, and will be resolved using the directory of the currently-executing Lua file (or / if such a dir cannot be found).

The path will also be canonicalised, i.e., use of . and .. will be removed. An error is raised if too many .. descend below the root of the game directory.

```
assert(`foo/../bar` == `bar`)
assert(`foo/../bar/.` == `bar`)
assert(`./foo/../bar` == `bar`)
assert(`/.` == `/`)
```

8.1.2. Unicode

All Lua files are now UTF-8. The string class in Lua has been rewritten to properly understand unicode. E.g. #"Â£" is 1, not 2. The most visible difference is that the syntax for regular expressions has changed. The new syntax is that of ICU's regular expression engine. For documentation, see sections "Regular Expression Metacharacters", "Regular Expression Operators" and "Replacement Text".

8.1.3. Vectors & Quaternions

Grit Lua scripts are often used to transform and manipulate objects in screen space and 3D space, as well as to manipulate colours. This is easiest with primitive vector (2d, 3d, 4d) and quaternion values. For performance reasons, we cannot use userdata or tables for our vector and quaternion values. This puts too much pressure on the garbage collector. Thus, the VM has been modified so that vector2,3,4, and quat are primitive types just like 'number', 'boolean' and 'nil'. They are copied by value and are not garbage collected. The following code shows some of the ways they can be used:

```
Q_ID = quat(1, 0, 0, 0)      -- constructing quaternions and 3d vectors
V_ID = vec(0, 0, 0)
V_NORTH = vec(0, 1, 0)
V_EAST  = vec(1, 0, 0)

local v = vec(1, 2, 3)
print(v)
print(v + v)
print(2 * v)
print(v * 2)
print(v / 2)
print(v * v)                    -- pointwise multiplication
print(dot(v, v))                -- dot product
print(norm(v))                  -- normalise
print(cross(V_NORTH, V_EAST))   -- cross product
print(-v)                       -- opposite direction, same length

print(Q_ID)

print(Q_ID * v2)                -- transform a vector

local q2 = quat(180, vector3(0, 0, 1))  -- angle/axis form
print(q2)
print(q2 * v2)  -- transform a vector by a quaternion

local q3 = Q_ID * q2 -- concatenate quaternion transformations
print(q3)
```

```
print(q3 * v2)
print(inv(q3) * q3 * v2)  -- invert a quaternion

print(v2 == v2)  -- equality is pointwise on the elements
print(q2 == q2)
print(q2 ~= q3)

print(#v2) -- length (pythagoras)
print(#q2) -- length (should be 1 for quaternions used to represent rotations)

print(unpack(v2)) -- explode into x, y, z
print(unpack(q2)) -- explode into w, x, y, z

print(quat(90, vector3(0,0,1)) * (V_NORTH + V_EAST)) -- angle/axis constructor form
print(quat(V_NORTH, V_EAST))                         -- quat between two direction vectors
print(quat(V_NORTH, V_EAST) * V_NORTH)

print(V_NORTH.yx) -- arbitrary swizzling is supported
```

8.1.4. Not A Number (NaN)

If you divide by zero in regular lua, you get a NaN value that goes on to pollute other values, until it eventually causes a problem in an unrelated area of code. This makes debugging difficult because it is hard to find the original cause of the NaN. In Grit we instead trap the divide by zero with an error instead of returning NaN.

8.1.5. Tail Call Optimisations

If the last statement of a Lua function is a call to another function (a tail call), the standard Lua compiler will perform an optimisation that results in a small performance improvement and allows unbounded tail call recursion (essentially allowing more programs to be written). Simply put, the interpreter re- uses the existing stack frame for the new call, which avoids using too many stack frames and therefore using too much memory. Here is an example of such a program:

```
function sum_1_to_n(x, sum)
    if x == 0 then return sum end
    sum = sum or 0
    return sum_1_to_n(x-1, sum+x)
end
```

Note that it is only a single class of recursive functions that can benefit from tail call optimisation. The following code implements the same function, using regular recursion. This code will create too many stack frames if given a high x input, as the tail call optimisation cannot be applied.

```
function sum_1_to_n(x)
    if x == 0 then return 0 end
    return x + sum_1_to_n(x-1)
end
```

Tail call optimisation removes stack frames, and thus removes lines from the stack trace if an error is generated. This can be highly confusing when debugging programs, as you can't find the line number where the bad call occurred.

It is not only in the case of these "tail recursive" functions where the loss of stack frame occurs, but in any function with a tail call. In almost all such cases the optimization does not provide any benefit at all.

We consider the small overall benefit of this feature to be a poor tradeoff for the productivity loss caused its effect on the readability of stack traces. Consequently, it is disabled in Grit. This means programs like the above must be rewritten as follows, to avoid using too many stack frames:

```
function sum_1_to_n(x)
    local counter
    for i=1, x do
        counter = counter + i
    end
    return counter
end
```

For an example of where lost debugging information causes a problem, see the following code, where the stack trace does not mention the critical filename or line number where the function was wrongly called. Imagine that there are many calls to do_something_tricky all over the program, and the arguments to the call are computed in each case. This means the stack trace is extremely useful for finding out exactly what went wrong, but the crucial stack frame is lost.

```
function do_something_tricky(x, y)
    if x ~= y % 3 then
        error "You fool, you called do_something_tricky wrongly."
    end
end

function i_am_buried_somewhere_in_a_million_lines_of_code ()
    x = 42
    y = 666
    return do_something_tricky(x, y) -- problem is here
end
```

8.2. Mathematics

8.2.1. Quaternion Basics

Quaternions are used to represent rotations in 3d space. They may seem scary, especially if you read wikipedia articles on the subject. The maths behind them is advanced, but you don't need to understand that in order to use them in Grit.

Quaternions are just a clockwise rotation of a given number of degrees about a given axis (think like screwing in a screw). Part of the reason they seem scary is that they are often given in 'raw' form like so:

```
quat(1,0,0,0)
```

These 4 floating point numbers are the in-memory representation of the rotation. They are labeled w, x, y, z, in that order. The fact that only 4 numbers are needed is the reason quaternions are popular. The most reasonable alternative, rotation matrixes, require 9 numbers. This particular quaternion is very important, as it is the identity quaternion. It represents a rotation of 0 degrees. When writing Lua code you can use the Q_ID global variable to make this more explicit.

It is possible to understand raw quaternions if you are reasonably good at mathematics, but it is much clearer to give quaternions in their angle/axis form. Here is the same quaternion in angle/axis form (you can choose any axis, since it is 0 degrees of rotation). These quaternion values all test equal:

```
quat(0, vec(0,0,1)) == Q_ID
quat(0, vec(1,0,0)) == Q_ID
```

A rotation of 0 degrees about any axis is not very useful. The following is a more interesting quaternion:

```
quat(90, vec(1,0,0))
```

This means a clockwise rotation of 90 degrees about the 'X' axis. To visualise this, imagine screwing a screw in the direction of the X axis. If you turn the screw 90 degrees, what used to point in the direction of the Y axis now points towards the Z axis. Likewise, what used to point in -Z now points towards Y.

To specify an anti-clockwise rotation about the X axis, one can invert either the angle or the axis. Inverting both results in an identical quaternion. The following are all equal:

```
quat(-90, vec(1,0,0))
quat(90, vec(-1,0,0))
quat(0.7071068, -0.7071068, 0, 0)
```

Generally, if you see a quaternion with 0.7 in the first value, it is a rotation by 90 degrees, whereas 0 indicates 180 degrees. In both cases, the last 3 numbers taken as a vector3 give the axis of rotation. Here are some example of 180 degree rotations:

```
quat(0, 1, 0, 0) -- 180 degrees about X
quat(0, 0, 1, 0) -- 180 degrees about Y
quat(0, 0, 0, 1) -- 180 degrees about Z
```

Note that there are many different ways of rotating by 180 degrees. E.g. a barrel roll, turning on the spot, and doing a backflip. You can in fact rotate by 180 degrees about any axis and the result is different in each case. On the other hand if you rotate by 0 degrees than it does not matter which axis you rotate around, because you are not rotating at all.

You can also get the quaternion that would transform one vector into another: The following quaternions all specify a rotation of 90 degrees about the 'X' axis. In the first case Y is turned to Z, and in the second case -Z is turned to X.

```
quat(vec(0,1,0), vec(0,0,1))
quat(vec(0,0,-1), vec(0,1,0))
```

It is possible to see the raw form of a quaternion by typing these other forms into the console. Also, one can easily extract the axis and angle from a quaternion using q.axis and q.angle.

8.2.2. Representing Orientations

The difference between a rotation and an orientation is that a rotation is a change in orientation. It is like the difference between an absolute value and an offset. In order to use an offset as an absolute value, we must assume a base value from which to offset to the value we want. In the case of simple numbers and positions, it is clear that the base value should be 0, or vec(0,0,0) respectively. However with rotations, one uses the identity quaternion, or quat(1,0,0,0) as a base.

For models, no rotation at all just means that XYZ in model space point in the same directions as XYZ in world space. However for lights, an unrotated light will shine in the +Y direction. Thus if you want it to shine in the -Z direction then it needs to be rotated by the following quaternion:

```
quat(vec(0,1,0), vec(0,0,-1))
```

Having lights shine in +Y by default is an arbitrary decision. Other applications instead prefer +Z, -Z, etc., so one must be careful when converting lighting setups from other software into Grit.

8.2.3. Transforming Vectors With Quaternions

If you have a point (or a direction, in the form of a direction vector) in model space, and you want to transform it into world space, you use the orientation of the object to rotate that vector. For example if you want to know if a car is upside down, you can trotate the up vector for the car in model space, vec(0,0,1), by the car's orientation, and test if it is still pointing up.

```
(car_orientation * vec(0,0,1)).z > 0
```

This operation can be chained. If you have a turret on a tank, and the turret usually faces +Y in model space, but you want to know where it points in world space given the turret angle (clockwise from the top) and tank orientation, you could use:

```
turret_point_model_space = quat(turret_angle, vec(0,0,-1)) * vec(0,1,0)
turret_point_world_space = tank_orientation * turret_point_model_space
```

You can also combine the two quaternions into a single one that describes the complete rotation:

```
turret_orientation_world_space = tank_orientation * quat(turret_angle, vec(0,0,-1))
turret_vector_world_space = turret_orientation_world_space * vec(0,1,0)
```

You can also invert a quaternion, which simply is the opposite rotation, i.e. the opposite angle around the same axis:

```
inv(quat(35, vec(0,1,0))) == quat(-35, vec(0,1,0))
```

This allows you to find out e.g. where the turret should point at in model space, in order for it point in a particular direction in world space. From there you can e.g. use math.atan2 to figure out the angle for the turret.

```
turret_point_model_space = inv(turret_orientation_world_space) * vector_to_enemy
```

8.3. Compilation Instructions

This is the central repository for the Grit Game Engine project.

From here can be built the engine executable itself, the launcher, and various tools. These are mostly useless without the accompanying media tree (the Game Directory which is available on Sourceforge via Subversion. Therefore to get everything, execute the following:

```
git clone --recursive https://github.com/sparkprime/grit-engine.git grit-engine
svn checkout https://svn.code.sf.net/p/gritengine/code/trunk grit-engine/media
```

The subversion tree also contains prebuilt up-to-date executables (Linux & Windows) so the majority of developers only need that. Grit can be substantially modified through Lua scripting, and this potential should be exhausted before modifying C++ code.

Build files are provided for Linux (`Makefile` and `.../*grit.mk`) and Visual Studio 2013 project files. Building C++ takes about an hour on Windows and 10 minutes on Linux. Scripts are available for copying new executables into Subversion, if it is checked out in the `media/` directory.

8.3.1. Windows

Only Visual Studio Express 2013 is supported. It is free (as in beer). Download it from the Microsoft site.

8.3.1.1. Requirements

You will need the DirectX9 SDK (Google it), install that on your system (in `Program Files`). The install adds a system-wide environment variable `DXSDK_DIR` pointing to the install directory. This is used by the Visual Studio build. If Visual studio is running, you will have to restart it to make it 'see' the new environment variable.

8.3.1.2. Regular build

Open grit-engine.sln and build the whole solution with the Normal configuration. This will build all the tools and dependencies.

8.3.1.3. Debug Build

Debugging with Visual Studio requires the engine to be built with the Debug configuration. To run in the debugger, execute the `engine` project from inside Visual Studio. You may need to set the working directory to the `media/` directory (from the `engine` project properties).

8.3.1.4. Modifying the Build

The build uses hand-written MSVC build files. Each executable and library has a project file, and properties files are used to layer additional build options without duplicating them between project files. They are structured as follows:

- `grit-engine.sln`: Collects together all the projects.
- `solution.props`: Build options for all libraries and executables. Options that are the same for both Debug and Normal configurations live here.
- `solution_debug.props`: Additional options when compiling in debug mode. Options that are the same for all object files live here.
- `solution_normal.props`: Additional options when compiling in normal mode. Options that are the same for all object files live here.
- `pch.props`: Options for enabling the precompiled header, used for top-level apps.
- `path/to/my-project/my-project.vcxproj`: An executable or library to build. Build options that are specific to the library itself (like warning levels) live here.
- `path/to/my-project/my-project.props`: Build options required by clients of a library and the library itself (typically defines and include paths).

8.3.2. Linux

The following instructions are for Ubuntu. If you're using another distro, you'll have to figure it out for yourself but hopefully the Ubuntu instructions will help. Note that the make files require GNU make, which may be called gmake on your system.

8.3.2.1. Requirements

```
sudo apt-get install subversion g++ make pkg-config gdb valgrind \
libfreeimage-dev libzzip-dev libfreetype6-dev libglu1-mesa-dev \
libxt-dev libxaw7-dev libglew1.5-dev libxrandr-dev \
libgoogle-perftools-dev libopenal-dev libreadline-dev freeglut3-dev \
nvidia-cg-toolkit libvorbis-dev xutils-dev libicu-dev
```

8.3.2.2. Building

Simply running `make -j 8` in the root (adjust for your number of cores) will build everything. Executables for the current platform are left in the root directory. You can add it to your PATH.

8.3.2.3. Debugging

You can debug Grit with `gdb` or `valgrind`. If the assembly is too obscure, disable optimizations by overriding the OPT variable as so:

```
make -j 8 OPT=
```

Note that this will not rebuild anything that is already built, so you might want to first delete specific object files -- the ones containing the code you're debugging, and then rebuilding them without optimizations.

8.3.2.4. Modifying the Build

The makefiles are handwritten. They use a lot of GNU make specific features. Each sub-project and dependency defines a `grit.mk` file which defines the additional source files and required build options. These are all included by the root `Makefile` which computes the actual build rules.