FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

PURGE

Design and Implementation of a High-Level Graphics Engine

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Necdet Can Atesman
Matrikelnummer 0004289

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: a.o.Univ.Prof. DI Dr.tech. Franz Puntigam

Wien, 05.09.2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

PURGE

Design and Implementation of a High-Level Graphics Engine

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Necdet Can Atesman
Registration Number 0004289

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: a.o.Univ.Prof. DI Dr.tech. Franz Puntigam

Vienna, 05.09.2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Necdet Can Atesman
Hietzinger Kai 71-73/9, 1130 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Abstract

When referring to the quality of 3D graphics engines, the central point of interest is usually computational performance. This very intense occupation with this aspect of such graphics engines leads to the exposure of performance-relevant implementation specifics in the API, to allow performance-aware usage of the product, but limiting the audience to those knowledgeable about the details of graphics engine implementation.

Our aim was to evaluate whether the amount of domain knowledge in existing graphics engine APIs was justified; whether it would be possible to reduce the knowledge expected from the user of such APIs without sacrificing too much performance. We have analyzed three different open source graphics engines to identify elements that are less known outside of the graphics development domain. Based on our findings, we have created PURGE, our own high-level API for graphics development, which makes use of other libraries for the actual rendering.

We have found that the performance impact of the added software layer was not noticeable during the development of simple test scenes, which we assume to be the prior concern of developers who are starting graphics development for the first time.

Zusammenfassung

Beim Entwickeln von Grafik-Engines liegt des Augenmerk meist auf der Performance, gemessen an der Anzahl Frames pro Sekunde, die die Engine zu generieren vermag. Dieser starke Fokus auf die Effizienz führt in vielen Fällen dazu, dass sich auch performancerelevante Details der Implementierung in der API wiederfinden um eine möglichst effiziente Verwendeung des Produkts zu ermöglichen. Durch diese Herangehensweise wird für die Verwendung solcher APIs einiges an Wissen in Teilbereichen der Grafikprogrammierung vorausgesetzt.

Wir wollen im Rahmen dieser Arbeit die Notwendigkeit dieser Herangehensweise überprüfen. Wir wollen herausfinden, ob eine Reduktion des benötigten Fachwissens einen direkten Performance-Verlust zur Folge hat. Wir haben zu diesem Zweck drei quelloffene Grafik-Engines analysiert, um Konzepte zu identifizieren, die außerhalb der Grafikdomäne wenig Verbreitung finden. Auf Basis unserer Erkenntnisse haben wir PURGE – eine eigene Grafik-API – entwickelt, die auf die Verwendung durch Entwickler ausgerichtet ist, die wenig bis überhaupt keine Erfahrung in dieser Domäne besitzen.

Anschließende Tests haben ergeben, dass diese zusätzliche Software-Ebene kaum Auswirkungen auf die Performance von visuell einfachen Applikationen hat.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Familiarizing oneself with a new domain requires one to learn and understand new concepts. In the case of 3D graphics engines[1], this means learning how a graphics engine works on the inside. The level of abstraction provided by many such library APIs is enough for developers with existing knowledge of this domain to adapt to a new API, but we found many of the libraries to provide an insufficient API for those without prior exposure to graphics development.

Graphical computation is a vast domain, where library-developers focus on the performance to achieve better-looking and/or more complex scenes. This very intense occupation with this aspect of such graphics engines leads to the exposure of performance-relevant implementation specifics in the API, to allow performance-aware usage of the final product, but limiting the audience to those knowledgeable about the details of graphic engine implementation.

Since advances in graphics hardware have made it possible for a modern graphics engine to supply several orders of magnitude more performance over the last decades, another approach to designing APIs for such engines will be explored in this document - one that focuses on ease of use, possibly sacrificing performance for this goal.

In this context, the aim of this thesis is to create an API that is easy to grasp for developers without specific knowledge of graphics engine implementation and usage. The resulting graphics engine called PURGE[2] is expected to handle as many unspecified-but-necessary parameters as possible with default values that try to predict the developers expectation. This behavior will most certainly lead to sub-optimal results, but permits fast draft implementations.

[1]The term *graphics engine* is used for any software library providing an API for graphical computation with a visual output.

[2]PURGE is an acronym for **PUR**istic **G**raphics **E**ngine. The library has no hard dependencies to other libraries.

The question is: is it possible to create a 3D graphics engine that can be used by developers that have no prior experience developing graphics applications. And what impact does it have on the usability, readability and performance of the applications making use of such an API?

With these premises, the API presented in this thesis supports fast development, with the option of optimizing the resulting application at the end of a development cycle as propagated by [17].

1.2 Approach

Since modern graphics engines cover a large range of topics, we will first select a sub-set of features of common graphics engines that we will support in our library. To compose this list of features, we will need a set of use cases which should cover the most common usages of the API when starting a project from scratch. These use cases will be implemented as tests at the end of the project.

In the next step, we will create a list of graphics engines to be used as reference implementations throughout the design process. This list of *reference engines* will cover projects with different design philosophies and project goals to avoid a homogeneous basis for comparisons. The selected engines must provide the complete feature list composed in the previous step.

We will then analyze the implementations of the chosen features in those reference engines and come up with initial architectural ideas. Those first attempts at the design will be refined in further chapters until we have a working implementation.

In order to obtain a fair base for benchmark comparisons, the library will not include any rendering by itself. All rendering operations will instead be delegated to other graphics engines by the PURGE library. The engine that is responsible for the rendering process (called `Renderer` in the final API) will then be integrated at run-time. Although this part of the application is interchangeable - there could be several different renderers using different graphics engines - we will only be using Ogre3d for this purpose.

We will eventually evaluate the usability of the resulting API using a number of test scenes. The scenes will be implemented twice - once with and once without usage of the API. The improvement will be measured by the lines of code, the number of function calls, as well as the number of *distinct* function calls to the API, as some operations might require multiple calls that could be reduced to a single one. Furthermore, a performance benchmark will disclose the impact of the added software layer on the test scenes.

CHAPTER 2

Related Work

There is plenty of literature available for the design and implementation of Graphics engines. None of the literature we have reviewed had information on the design of the API. It is much more likely to find generic information about programming [10], object-oriented development [9], dynamic linking [36] and coding guidelines [8].

In order to discuss the API of graphics engines, we will first have to look at the components of a graphics engine. There are many aspects of a graphics engine that need to be addressed when developing such a library. A very good overview to this subject is provided by [3]. As our focus is on the API of the introductory articles, we will merely cover the API required by those tutorials.

The very first applications built when learning graphics development involve the programmatic generation of a scene. The developer usually learns these few operations at first:

- Placing objects in a 3d space,
- moving/rotating those objects to other positions,
- moving/rotating the camera to view the scene from different angles, and
- performing the repositioning operations discretely within a time frame.

2.1 Scene graph

This data structure has become the de-facto standard for storing objects in graphics engines. Paffenroth et al. [26] describe it as a “well-known and longstanding notion in computer graphics, either as a structural hierarchy of the view scene [6], or a spatial development of it [33] [34].”

A *Scene Graph* usually is a direct acyclic graph containing objects in the scene [32]. Each node within the tree has several properties (like location, orientation and scale) that usually propagate to child nodes. Placing a dinner table in a room could look as depicted in Figure 2.1.

The objects "on" the table have a relative position referencing the position of the parent node. This solution allows us to reposition the complete structure without having to reposition each sub-node separately.

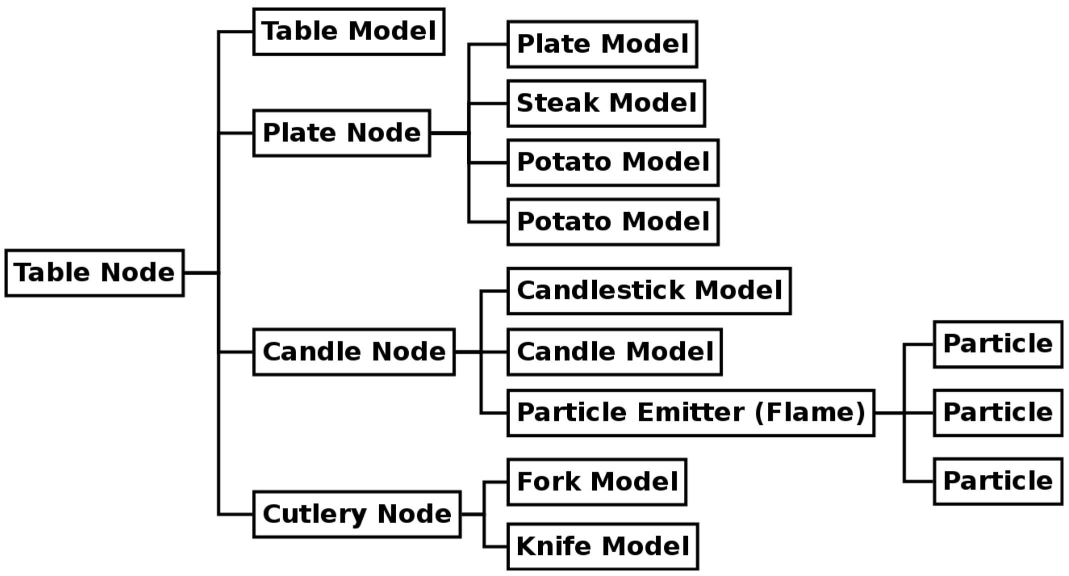


Figure 2.1: An example scene graph depicting a dinner table

Apart from these usability features, scene graphs are further useful for the calculation of bounding volumes, which in turn are used for clipping. Assarson et. al summarize several such techniques and shows a speed improvement of up to 11-fold using these methods [1].

2.2 Quaternions

Positioning an object within a scene graph requires the expression of its orientation relative to its parent node. There are several methods for expressing the attitude of an object within 3-dimensional space, which are summarized in [7]. The *quaternion* has become the de-facto standard for this purpose in graphics engines [24], as it is neither as memory-intensive as a transformation matrix, nor prone to the so-called gimbal lock as Euler angles are.

Quaternions are hyper-complex numbers of rank 4, constituting a four dimensional vector space over the field of real numbers [4]. A Quaternion is expressed as a four tuple

$$q = (w, x, y, z) = w + ix + jy + kz$$

which consists of a vector part x, y, z, the scalar part w and where the units i, j, k satisfy:

$$i^2 = j^2 = k^2 = ijk = -1 \tag{2.1}$$

$$ij = -ji = k \tag{2.2}$$

$$jk = -kj = i \tag{2.3}$$

$$ki = -ik = j \tag{2.4}$$

A quaternion expresses an attitude within 3d-space – more precisely the rotation towards an attitude. The multiplication of quaternions $(A \cdot B)$ correspond to sequential application of their rotations and yields a new quaternion (C) which expresses the first rotation followed by the second rotation. As the first rotation will rotate the axes of the second rotation, changing the order of the multiplicants results in a different compound rotation, rendering the quaternion multiplication non-commutative. This property of rotations is visualized in Figure 2.2.

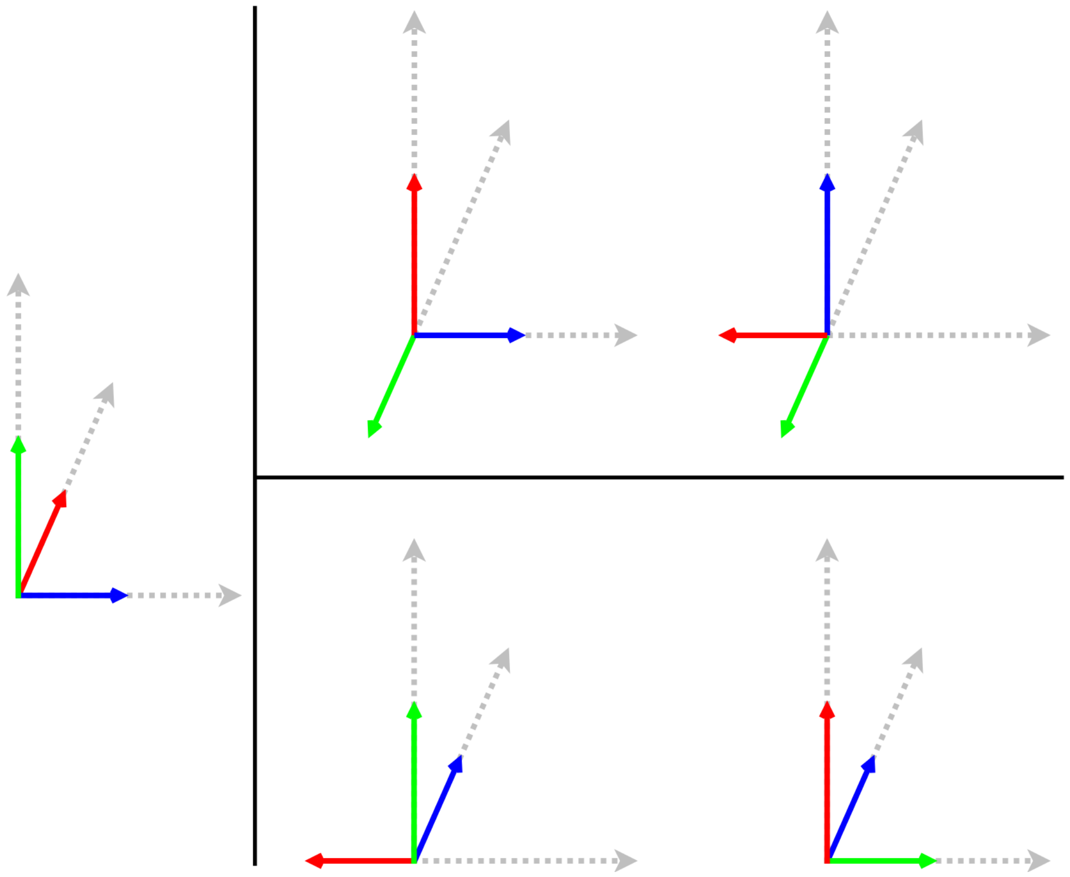


Figure 2.2: Changing the order of rotations yields different results: The rotations above (90 ° around blue, 90 ° around green) are applied to the original orientation in reverse order in the lower half, resulting in different final attitudes.

2.3 Render Loop

Once the initial setup of a scene is completed, it needs to be drawn onto the screen each time an object in the scene is modified. Instead of rendering the output each time the scene is updated, most graphics engines choose to redraw it in an infinite loop, assuming that the output will change frequently enough that a constant redraw is necessary. This behavior is in contrast to that of window managers for example, which usually redraw the screen only when necessary.

There are several aspects to the design of this infinite loop. Valente et al. [35] list and categorize several such models. The most simple implementation, the “simple coupled model”, is visualized in Figure 2.3. The objects are updated with fixed values in this model – the camera rotates 1 ° each frame, for example.

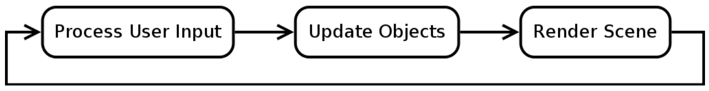


Figure 2.3: The most simple render loop

The disadvantage of this simple design is its dynamic frame rate – some operations will take longer than others, resulting in greatly varying intervals between single frames, which will result in different results on different hardware. Synchronizing the output of the renderer to a fixed interval is achieved by a simple blocking operation at the end of each cycle as seen in Figure 2.4. This will guarantee an output at a fixed rate, like 30 frames/second, and allows the developer to use constant values during updates as with the previous model.

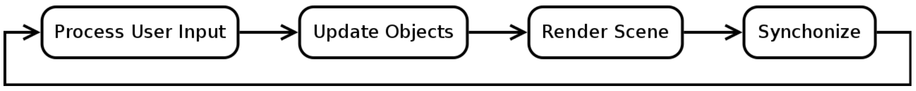


Figure 2.4: A simple render loop with synchronization.

This approach is still insufficient, as the application starts to stutter whenever a cycle needs more time to finish than the predefined time slot. The solution is to perform all modifications to the objects in the scene using the time elapsed since the last cycle. This is the “Single-thread Uncoupled” model shown in Figure 2.5. It is further possible to add a synchronization to the end of the loop, limiting the required operations to fixed multiples of the constants required during the loop. If a cycle needs two time slots in the example above (i.e. more than 1/30, but less than 2/30 seconds), the next cycle will rotate the camera by 2 ° instead of 1 °.

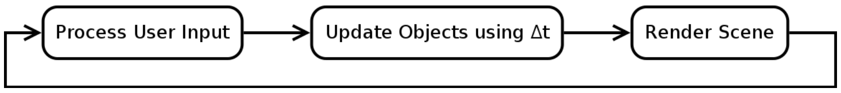


Figure 2.5: Single-thread Uncoupled model.

Other models operate on multiple threads. Most importantly the rendering is performed in a separate thread, which blocks until the world updates in other threads are in a consistent state. These approaches make use of the multiple CPU cores present in today’s hardware. Another big advantage of such models is that the rendering itself is performed on the GPU, while the other operations are usually bound to the CPU. All previously presented models stress either of these resources, while the other stays idle.

It is possible to tackle this issue without the need for threading when using double buffering – a common feature of modern 3d graphics cards. Most calls to OpenGL will return immediately while processing the issued command on the GPU. The call to the OpenGL function `SwapBuffers()` – which flushes the rendered image onto the screen – will block until all pending operations are finished. This makes it possible to perform other operations during this time frame. This approach is outlined in Figure 2.6.

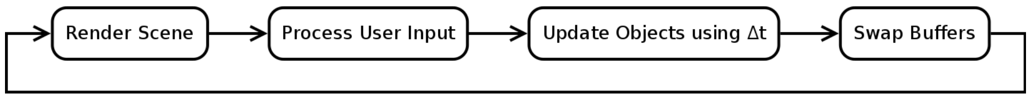


Figure 2.6: A render loop in OpenGL performing operations in parallel on the CPU and the GPU.

2.4 Models

A *model* is a 3D object that can be positioned in space. Usually models have a predefined shape, a *mesh*, consisting of a collection of triangle definitions. The mesh alone is not enough to render any realistic scene, additional *material* definitions are attached to triangle groups which define how the triangles are to be drawn by the renderer. These materials are implemented by dedicated code fragments that run on the GPU. These self-contained applications – so-called *shaders* – are categorized by the objects they operate on [3].

- *Vertex Shader*s operate on single vertices – the points in the 3D-space of the mesh forming the triangles – of the mesh. This shader type receives a single point as input and must operate on that vertex alone without any knowledge of its surroundings.
- *Geometry Shader*s receive multiple vertices that were processed by the vertex shaders. They have the ability to add new vertices or remove/modify existing vertices.
- *Pixel Shader*s (also called *Fragment Shader*s) operate on the pixels of the final image, just before it is drawn onto the screen. Figure 2.7 displays a few examples of pixel shaders.

As all of these shaders operate on tiny fragments of the complete scene, the application of the shaders can be run in parallel. Modern graphics cards can run hundreds of operations in parallel for this purpose [25].

2.5 Lighting

Although lights are usually defined as self-contained entities in a scene, they are never drawn directly. Instead, they define how other objects in the scene are rendered. Heidrich et. al. summarize various lighting models in virtual environments in [14]. A light source usually consists of two components: diffuse and specular.

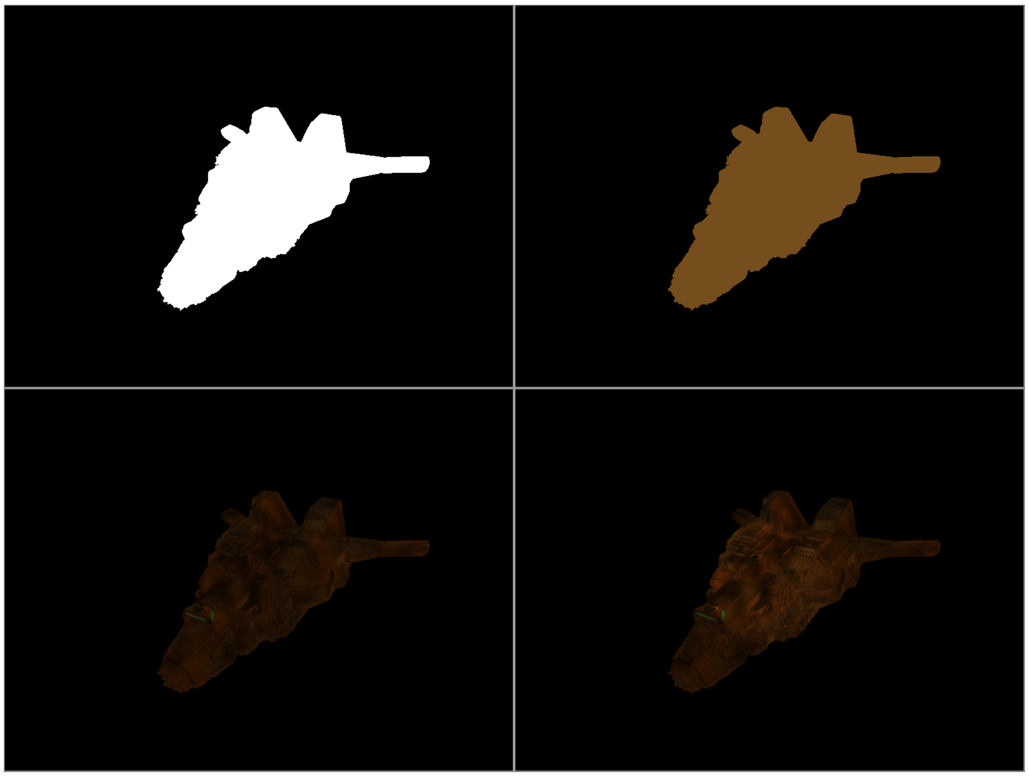


Figure 2.7: Examples of pixel shaders: the same model a.) without any shaders, b.) with a uniform color, c.) with a texture and d.) a shader rendering a spotlight onto it.

- The diffuse component of a light ray is scattered at many angles upon hitting a surface and can thus be seen from various angles, whereas
- the specular component is reflected at the same angle at which it hits a surface.

As light sources are only observable by the illumination of the objects in the scene, these two components are defined on the material of all surfaces, too. Whenever a diffuse (or specular) light would illuminate a surface, the material of the surface defines the effect of the illumination. A mirror will almost ignore the diffuse component of the light, but reflect the specular component with the same intensity as the incoming light beam. Most natural fabrics will do the exact opposite.

Another property of lights is the direction the light beams are emitted towards. The common categories defined by this behavior are:

- *point light*s – unidirectional light sources,
- *spot light*s – emitting light in a cone,
- *directional light*s – coming from an infinitely distant point, giving off parallel rays – and
- a single *ambient light* – uniformly illuminating everything equally from all directions.

The last parameter for light sources is their attenuation behavior – how the light emitted from a light source is reduced depending on the distance to the surface it illuminates. This is actually rather a property of the environment in which the light source operates, as light does not fade as long as it travels unhindered in vacuum. A dusty room, on the other hand, would cause light sources to be less effective with increasing distance. The attenuation is considered a property of the light source nonetheless for technical reasons: The modification of illumination based on the properties of the sub-spaces between the surface and light source would be quite complex and computationally extremely expensive.

2.6 Cameras

A camera is a viewpoint from which the scene is rendered. The visible space to a camera is the so-called *view frustum* [20], and is constrained by the near and far clip distances as seen in Figure 2.8. Solely objects within – or reaching into – this area are rendered when viewing the scene from this camera.

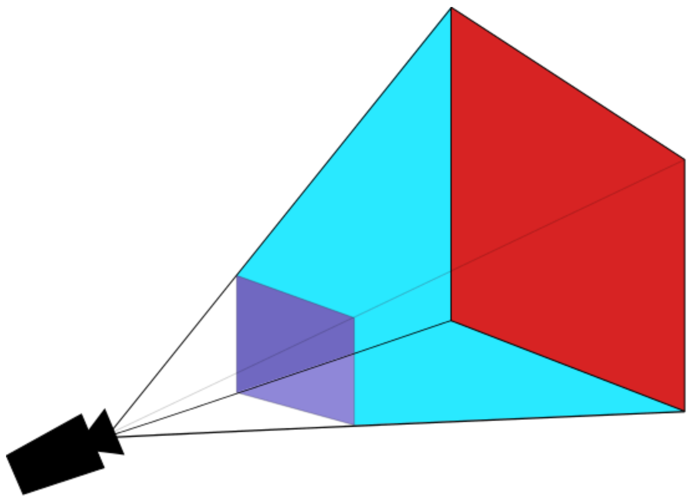


Figure 2.8: The view frustum of a camera, contained by the *near clip distance* in purple and the *far clip distance* in red.

The other two parameters for the definition of a camera are its field of view – the horizontal angle of the frustum – and the aspect ratio defining the vertical angle. It is possible to establish further properties that simulate the effects of different camera lenses [18], but we will limit our model to the simplified definition above.

CHAPTER 3

Scope

3.1 Reference Engines

Throughout the thesis, we will have other graphics engines as reference implementations and API examples. We have chosen three different engines with varying aims as such reference engines:

- OpenSceneGraph: This engine features a very powerful but complex scene graph implementation. The complete design philosophy is focused on the scene graph, the data structure managing the objects within the scene. This data structure is very important in modern graphics engines and will be explained a bit more in the next chapter. The API of the engine itself is targeted at developers with experience in the graphics domain and it is quite complex, requiring good knowledge of the math behind the scenes. An introduction to the engine can be found in [19].
- Panda3d: Initially produced by Disney's VR Studio, this engine has become an open source project that is still used in commercial projects of Disney[1]. The engine itself is developed using C++, but the engine comes with a python interface allowing rapid prototyping without the need for recompilation [13]. We have found that it has indeed a very clean and simple API that allows the creation of simple scenes with few commands.
- Ogre3d: A quite popular open source graphics engine that was used in several commercial games. According to its web page, it was "designed to make it easier and more intuitive for developers to produce applications utilizing hardware-accelerated 3D graphics"[2]. Although the API is quite verbose, it has a very clean architecture and good documentation.

[1]*Toontown* (http://toontown.go.com/) and *Pirates of the Caribbean online* (http://piratesonline.go.com/welcome) being two examples

[2]http://www.ogre3d.org/about

3.2 Features

In order to keep the focus on the architectural design within the engine, we will implement a very limited feature set. We will use the topics covered by some introductory books on graphics engine usage to determine these features: [31], [16], [23], [9] and [22]. All five books are targeted at developers without prior knowledge of graphics engines and four of them introduce the reader to the API of one specific graphics engine. Although all of them cover many more topics, the common areas can be summarized as:

- scene graph manipulation,
- camera manipulation,
- lighting,
- loading and manipulating objects (including meshes, materials, textures),
- operations on a 2-dimensional space,
- processing user input and
- usage of the engine within a render loop.

Other areas found in more than one of these books are:

- picking objects,
- particle effects,
- loading terrain,
- collision detection and
- visual post-processing.

Having a list of common topics in introductory books, we can now pick a set of features that will be included in PURGE.

Renderer

The most important decision to make first is the low-level graphics API to use. We could make use of OpenGL or DirectX, but this approach would not provide a good base for benchmark comparisons, as the rendering process is extremely complex and the required optimizations would cover much more ground than the API itself.

Instead, we will be using the API of other graphic engines. PURGE will solely handle the high-level scene layout and delegate all rendering operations to another, external renderer. This approach will allow us not only to concentrate on the API, but will further provide the ability to compare performance metrics of the resulting library to those of the unmodified renderer.

Details on the design of this architecture are provided in Section 4.1, whereas the implementation details can be found in Section 5.2.

Scene Management

First, it is important to note that all covered engines make use of scene graphs as a means of managing objects in 3-dimensional space. We will need to define a set of operations we want to support on the nodes of this scene graph. The common properties of the above graphics engines are

- location,
- rotation and
- scale.

Changing the *Location* of an object "moves" the object by a given amount. The *Rotation* does not change its location per se, but moves all points of an object around an axis in a circular manner.

The *Scale* of an object is defined as adjusting the distance of each point of an object to a given point of reference by a scalar. The scale is defined separately for each coordinate axis and can be negative. The complete scale can thus be defined by a tuple of three scalars, one for the scale of each coordinate axis. This implicitly defines a mirroring feature: Scaling by $(1|1|-1)$ results in the object being everted along the Z-axis.

Lights

Lighting is the feature that affects the realism of a scene the most. As implementing a decent lighting model would go way beyond the scope of this thesis, we will use an extremely simplified model. The light sources in the final library will be defined by

- a specular color,
- a diffuse color and
- a linear attenuation value.

These parameters define the "behavior" of light rays which are emitted from a light source. Another component of light sources is the direction these rays are emitted towards. The types of light present in most engines are

- point lights,
- spot lights,
- directional light and
- a single ambient light.

The first two light sources – point and spot lights – can be integrated into the scene as tangible objects. They can be positioned, rotated and scaled as defined by the scene node operations above. Although rotating a point light has no visual effect on the output, it is possible that the rotation has an effect on other objects that are attached to this light source, as we will discuss in Section 4.5.

The other two light sources – directional and ambient lights – are not integrated into the scene graph, but rather defined outside of it as global illumination parameters. As these light sources do not have a single point as origin, they do not have any attenuation either. The sun is a good example of a directional light source that emits light in a single perceived direction, and has no significant attenuation. The ambient light can be thought of as the illumination resulting from diffuse reflections within the scene.

The design of the light classes can be found in Section 4.5.

Models

The final area to be covered is the loading of models into the scene. Although every graphics engine supports various formats, each of them supports reading a geometric model from external resources - most commonly files. Our reference engines are not limited to models when loading entities from files. They have own scripting languages or exporters that can be used to define other object types that can be loaded into the scene. These entities are not necessarily handled by the engine, they might as well be processed by plug-ins. Some examples include

- light sources,
- billboard effects,
- particle effects,
- dynamically generated models (trees, for example),
- terrain and
- sky boxes.

To cope with this variety of entities, the object model of PURGE will not distinguish effects from predefined or dynamically generated models. All loadable objects are treated equally. This decision has one important implication: The resulting library will not be able to manipulate these objects. It is capable of triggering the loading of an object by its name, but incapable of altering an object that was loaded this way. This fact further implies that light sources integrated as "models" are indistinguishable from other objects and cannot be modified at a later time.

3.3 Excluded Features

To further define the boundaries of the engine, we will establish another list of explicitly excluded features; especially since several features not covered in this thesis might be perceived as inherent features of a graphics engine.

- Rendering to anything except the screen: Rendering to the screen is assumed to be the primary concern of a developer making use of such a simplistic API,

- 2d objects: Neither the rendering of menus, nor any other two-dimensional overlays are covered in this thesis. The focus will be solely on the management of the three-dimensional scene.

3.4 Use Cases

As the target audience consists of software developers with little or no prior experience with graphics engines, the library will provide the most convenient API for the simplest tasks. We considered the following list to be the most rudimentary tasks when confronted with a graphics engine API for the first time:

- Creating an empty scene and looking into that scene through a camera: This step involves the creation of all necessary objects to enter the rendering loop. As this step is part of every application, the reduction of this boilerplate code could be considered to be the most important use case.
- Positioning an object in the scene: Loading an object into the scene and altering its coordinates, orientation and/or scale to be visible through the default camera.
- Updating an object: Moving, rotating or otherwise modifying a previously loaded object discretely in a pre-defined time period.
- Controlling the camera: Updating properties of the camera, enabling the implementation of a dynamic scene.

This list of use cases will be used during the evaluation of the final architecture in Section 6.

CHAPTER 4

Design

Since the aim of this thesis is to provide an API usable without domain-specific knowledge of computer graphics, the API will try to hide as many domain-specific aspects of graphics engine usage as possible. It will need to predict some intentions of its users. For example, the library will automatically create a render window if none was created by the user before the render loop was entered. More such details are summarized in Section 5.4.

It will further try not to hide away such implementation details completely, giving more experienced developers the level of control available in other graphics engines. This approach effectively creates an API that is usable on two levels:

- as a complex, feature-rich graphics engine, and
- as a simplified engine with many default parameters and operations.

The driving force behind any decisions during the design process was the principle of least astonishment [30]. The adopting developers are assumed to be developers that have very little or no prior knowledge of implementing graphics applications, which implies that some behaviors of the resulting engine might still be unexpected to those knowledgeable in this domain. An experienced developer might want to enter the render loop without any active render windows, for example.

To support these operations, all automatic behavior will be modeled to be suppressible. It is still possible to start the main rendering loop without any target to render to if this is the actual intent. But, where other engines would insist on a choice – and would probably abort execution due to a lack thereof – PURGE instead tries to make that choice for the developer. This naturally means that the choice might not be the right one for users familiar with this domain.

4.1 External Renderer

Luckily, the usage of an external renderer already provides access to an advanced API. Users knowledgeable about the external graphics engine being used can fine-tune operations within that engine. Any object controlling the rendering process is created within the PURGE namespace, and adopted by the external rendering facility, the `Renderer`. That facility is responsible for taking that object into account during the rendering process.

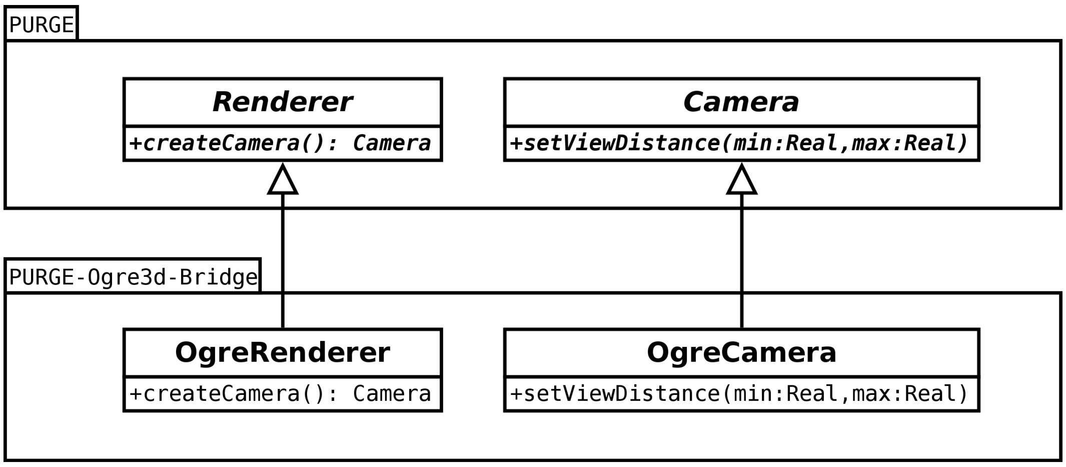


Figure 4.1: Initial design approach to the external renderer

The first approach to the integration of the external render was to design the `Renderer` as an abstract class that is responsible for providing all other objects supported by PURGE, as seen in Figure 4.1. The drawback of this architecture became clear quite early during the implementation process: The design of PURGE was adopting too many characteristics of the underlying renderer. The library was becoming a layer on top of Ogre3d instead of an independent graphics library.

To counter this trend during the design process, the architecture was slightly altered to decouple PURGE objects from their implementations in the renderer. The aim was to render the `Renderer` completely interchangeable, even at run-time, an approach to the engine design we have found in [28][1]. This has lead to the second design outlined in Figure 4.2, where the objects in PURGE can survive an exchange of the active `Renderer`. We switched from direct inheritance to the bridge pattern as described in [12].

But this change had some drawbacks: Although the PURGE object was now capable of storing its parameters, the API had not changed. The object would now store its state on its own, but immediately update the implementing object. Apart from this lack of solution to the initial problem, another, more grave design flaw was brought to our attention with this switch. Why should the library actually enforce the duplication of the already-present data in the `Renderer`?

[1]Plummer basically contemplates on a game architecture proposal in [29] and attempts to create a game engine with completely interchangeable components for specialized tasks.

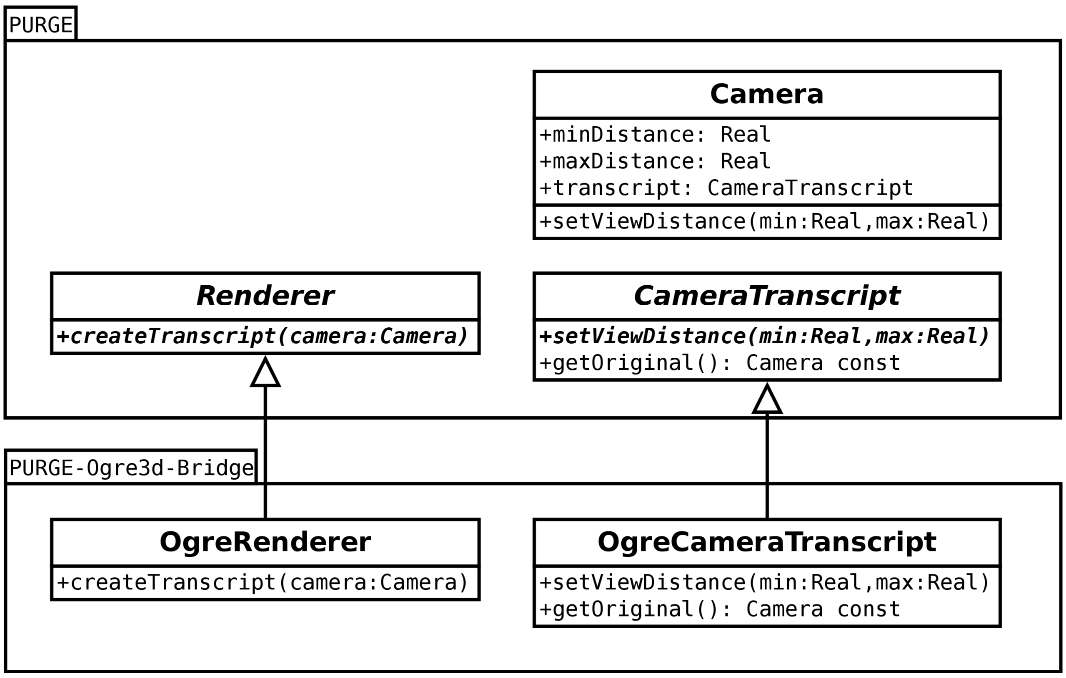

PURGE
Camera
+minDistance: Real
+maxDistance: Real
+transcript: CameraTranscript
+setViewDistance(min:Real,max:Real)
Renderer
+createTranscript(camera:Camera)
CameraTranscript
+setViewDistance(min:Real,max:Real)
+getOriginal(): Camera const
PURGE-Ogre3d-Bridge
OgreRenderer
+createTranscript(camera:Camera)
OgreCameraTranscript
+setViewDistance(min:Real,max:Real)
+getOriginal(): Camera const

Figure 4.2: Second design of the external renderer

What was the `Transcript` good for? It was a required implementation detail in the initial design, but the usage of the bridge pattern had made that class obsolete for PURGE. The `Renderer` was still free to create a mapping of a PURGE-object to its own, graphics-engine specific objects, but PURGE does not need any information from these objects.

So this short-lived solution was discarded and the flow of information was reversed: Instead of pushing any changes from PURGE to the `Renderer`, the `Renderer` is now expected to pull any information it needs from PURGE. The outline of this design can be seen in Figure 4.3.

With this last change, the flow of information is defined as follows:

1. The application can create, update and delete any number of objects in PURGE. None of the changes have any visible effect at this point.
2. When all changes for this loop cycle are declared to PURGE, the rendering step initiates.
3. The `OgreRenderer` is responsible for creating, updating and deleting all required internal objects to render the scene as contained in PURGE.
4. When the renderer is finished, control is passed back to PURGE, where a new loop cycle begins.

The implementation of this communication model requires further elaboration on the details of the render loop. We will come back to the implementation specifics in Section 5.2.

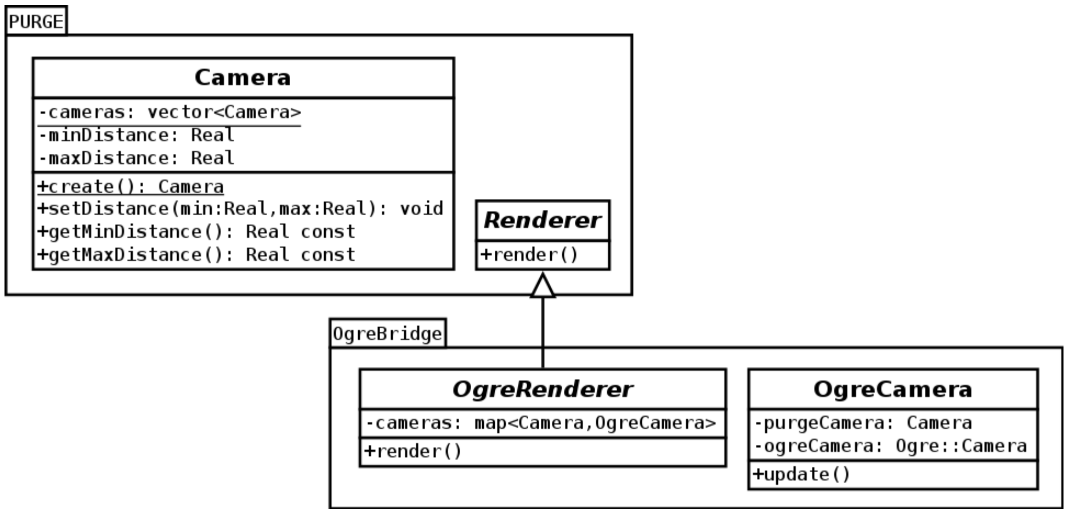


Figure 4.3: Third iteration of the renderer architecture

4.2 Render Loop

The communication between PURGE and the external `Renderer` is heavily dependent on the render loop – or main loop – of our library. The loop basically consists of three different stages that need to be run through repeatedly:

1. gathering user input,
2. updating the scene and
3. rendering the scene.

We will be using the "Single-thread Uncoupled model" presented by [35], which cycles through these stations periodically in a single thread and updates the objects using the time elapsed since the last cycle. We will see in Section 5.4, that the choice of the implementation does not have a strong impact on the API.

The render loop in a graphics engine must provide an API for the repeated modification of the scene by the application. The sequence described above (gather input, update, render) is present in all reference engines, but each of them approaches this problem quite differently:

- Ogre3d: Provides the FrameListener interface[2] for the registration of recurring tasks during the loop cycle. Developers can implement this interface to react to certain events in the rendering loop, usually the end of the render cycle. The current version provides three such events:
    - `frameStarted()`: Called before the frame is being rendered.

[2]http://www.ogre3d.org/docs/api/html/classOgre_1_1FrameListener.html

  - `frameRenderingQueued()`: Called while the GPU is computing the current frame. This event is useful in the case one wants to make use of the CPU at this time, as the CPU would otherwise be idle and the application would block until the GPU is finished.
  - `frameEnded()`: Called after the rendering is finished.

  Ogre3d does not have native routines for processing input from external devices (such as keyboards or mice), but an external project is recommended for this purpose throughout the documentation: OIS[3]. This external library again provides two methods of processing input: buffered and unbuffered. If the library is used in buffered mode, all input can be processed at once during the execution of a frame listener. In unbuffered mode, a listener has to be provided to the library that will be called immediately on the retrieval of new events through system interrupts.

- Panda3d: Allows the registration of tasks to be continuously performed at a central facility, the `AsyncTaskManager`. As the name suggests, the library is capable of distributing ongoing tasks across different threads and actually makes use of this mechanism to manage the rendering loop. The rendering step itself is nothing more than a `GenericAsyncTask`. A nice option is the registration of function pointers instead of full-blown classes, which makes playing around with the library a bit more convenient.

  The `AsyncTaskManager` holds multiple instances of `AsyncTaskChain`, which contain objects of type `AsyncTask`. Each `AsyncTaskChain` can run in a separate thread. Furthermore, each `AsyncTask` has a sort number that ensures that tasks can be executed in a predefined order across all `AsyncTaskChain`s. Figure 4.4 visualizes the sequence in a single cycle with two `AsyncTaskChain`s.

  Unfortunately, the name `Chain` is a bit misleading in this context. The name suggests a linear dependence between "links", which is not the case: The "links" in the chain have additional dependencies to other links in other chains, as seen in Figure 4.4. A more accurate visual representation of this approach would possibly be "thread", reminding of the computational thread it possibly runs in and the thread of a woven fabric, having dependences to other threads.

- OpenSceneGraph: As the project does not provide a render loop *per se*, it is possible to implement ones own loop that takes care of any operations in-between the rendered frames.

  But another peculiarity of OpenSceneGraph is the strong focus on the scene graph, even when designing the application flow. We had seen that the scene graph is actually a data structure for organizing objects in space. OpenSceneGraph additionally allows the registration of callbacks to scene nodes. These `NodeCallbacks`[4] are called during the rendering process as part of the render loop that updates the scene graph.

[3]The official web-site of the project is http://www.wreckedgames.com, but more information can be found on its sourceforge project site: http://sourceforge.net/projects/wgois/

[4]http://www.openscenegraph.org/documentation/NPSTutorials/osgFollowMe.htm

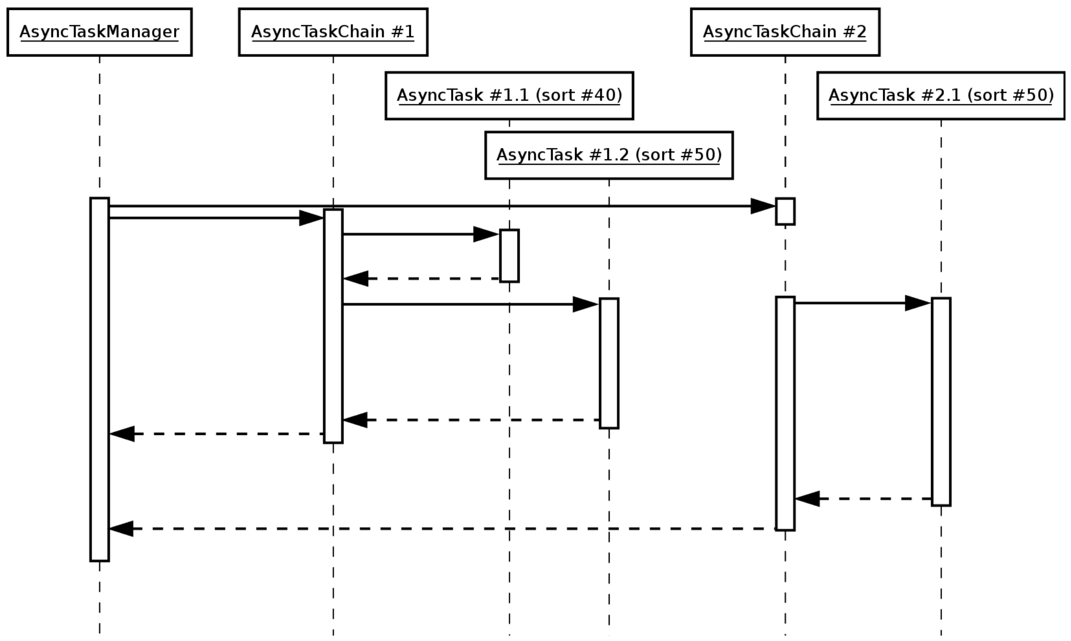

AsyncTaskManager
AsyncTaskChain #1
AsyncTaskChain #2
AsyncTask #1.1 (sort #40)
AsyncTask #2.1 (sort #50)
AsyncTask #1.2 (sort #50)

Figure 4.4: Simplified sequence diagram of a single rendering cycle in Panda3d. `AsyncTaskChain #1` and `AsyncTaskChain #2` start running simultaneously, but the second chain will block until the first task in the first chain was executed, as it has a smaller sort value than the task in the second chain.

Among the analyzed libraries, Panda3d provides the most flexible API for managing the scene during the main loop. It allows us to split the loop into multiple threads, making good use of multiple CPUs, but the API emphasizes an early decision which tasks could be performed in separate threads. It requires the definition of `AsyncTaskChain`s for the integration of new tasks into the loop. A dependence between the tasks is then established through the so-called sort numbers.

An alternative implementation could hide away the details of threading just by introducing Task groups as outlined in Figure 4.5. A Task group adopts the duty of the numeric sort number in Panda3d's loop design. Instead of assigning a value to a task, it is instead added to the same task group as other tasks that can be processed simultaneously.

This implementation is in contrast to Panda3d's implementation choice, as it does not group tasks by their ability to run in parallel, but rather creates groups of tasks that need to be performed sequentially – handling input, computing Physics, computing AI choices, updating positions, and rendering are some examples of such task groups.

The design described above is equivalent to having a single flat rendering loop that executes tasks sequentially. The `TaskGroup` structure is introduced for convenient re-arranging of the loop elements as a tree structure. With an adequate `TaskGroup` re-implementation it would be further possible to insert parallelism into this design. Although this implementation is not accomplished within this thesis, we will look at it briefly in Section 5.4.

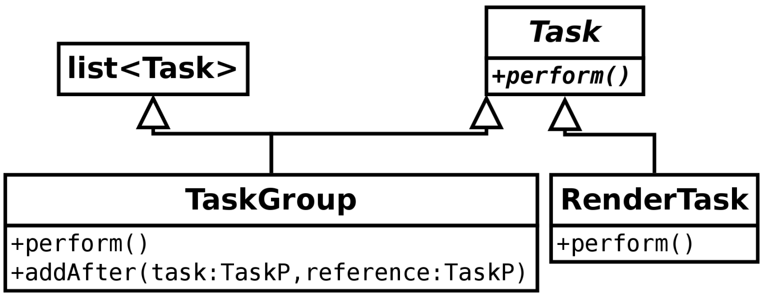


Figure 4.5: `Task` and `TaskGroup` class hierarchy.

To further simplify the process of adding Tasks to a task group, we will add additional functions to the `TaskGroup` class. These functions will further improve the usability by reducing the number of calls necessary to perform common operations: Adding a task after another reference task, for example.

While looking into the implementation of the communication between PURGE and the `Renderer` in Section 5.2, we will further discuss the single mandatory task of the library: the `RenderTask`.

4.3 Cameras and Windows

We will implement a simplified model of cameras, defined by near- and far clip distances, as well as a field of vision. The aspect ratio will always match that of the render window, preventing accidental distortion of the render output.

A window is a 2d-plane where the view of a camera can be projected to. As the window is partially managed by the system's window manager and modified by the user of an application, this entity should additionally be regarded as a communication device. The user can move, re-size or close the window. We will only model a single such interaction to prove that the architecture could support others as well. Window objects in our library will be able to detect when the user has closed a window.

All our reference engines allow us to partition the render plane of the windows to create multiple smaller regions that each display the rendered output of one camera, as demonstrated by Figure 4.6. This approach allows developers to implement split-screen games, for example. All three engines provide this feature, although they all use different names for this fraction of the window displaying the output of a camera:

- Ogre3d calls them *Viewports*,
- in Panda3d, these are *DisplayRegions*, whereas
- OpenScenegraph refers to them as *Views*.

We will be using the term `Viewport`, as this is the technical term for this construct and the other names do not provide any additional insight into this entity. Figure 4.7 shows the general

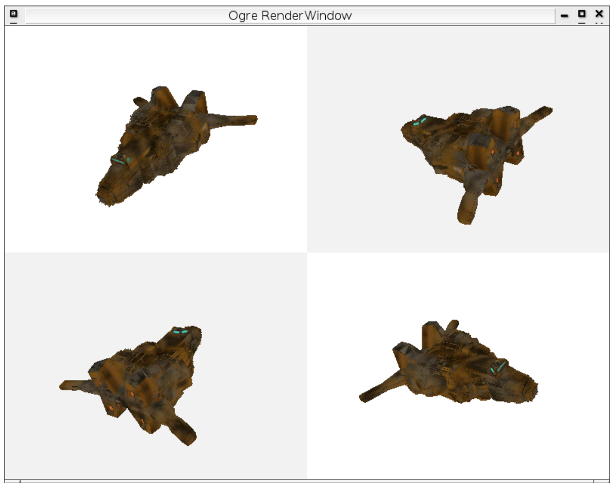


Figure 4.6: A render window displaying a single space ship in four different viewports.

association between windows, viewports and cameras[5]. A viewport contains information on how the camera view is embedded into a window.

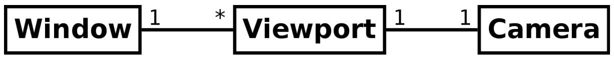


Figure 4.7: Associations between windows, viewports and cameras.

One important property a viewport needs is its *z-index*. This value is consulted whenever viewports overlap. The one with the lower value is drawn first and is overlapped by the viewports "above" it.

The implementation of these structures are described during the implementation of the `Renderer` in Section 5.2.

4.4 Relative Orientation

The Cartesian *Coordinate System* of the 3-dimensional space of a scene has two properties that need to be defined in order to position objects therein:

- Handedness: The direction the third vector points to, when the first two are defined.
- Orientation: The directions the three vectors point to relative to the render window.

The orientation is unnecessary for the definition of the coordinate system from a mathematical point of view, since all orientations with the same handedness can be transformed to each other using a single rotation around the origin according to Euler's rotation theorem [27]. But

[5]A viewport is actually not limited to a single camera in all reference implementations, but we will stick to this simple model for our library.

since human orientation rather happens in terms of relative directions – up, down, left, right, forward, backward – we will try to use these directions instead of the named axes X, Y and Z.

The lack of a standard mapping[6] is taken as an indicator that it is best left to personal preference. The choice will not have any effect on the API if orientation is expressed throughout the application, but it will still have two effects on the result:

- If relative movement using the terms “up” or “down” are used alongside vectors for positioning objects, the three components of the vector will each map to one relative direction. There will be a visible difference between the exact same applications using different coordinate systems.
- The other implication is not so obvious: The underlying renderer will need to perform additional operations if the handedness of the coordinate systems differs from its own usage. We will elaborate on this effect during the implementation of the `Renderer` in Section 5.2.

Conceptually, the desired orientation serves but a single purpose: to assign a default rotation to newly created objects. All rotatable objects have a predefined orientation, or predefined directions that are considered that object’s forward, right and up vectors. The orientation of the coordinate system rotates all new objects to align those directional vectors to those of the coordinate system’s orientation. Figure 4.8 shows a newly created spot light that was positioned facing forward in the coordinate system orientation of Direct3D9 – assuming that the forward orientation of a spot light is defined by the direction vector its light is emitted to.

As long as the coordinate system to use during development is defined before any objects have been created, the library does not need to care about the coordinate system in use. We will need to define how the application behaves if the coordinate system is changed at run-time:

Although it would technically be possible to re-position all objects within the scene, we have chosen to ignore this event. Although changing the coordinate system after having used it for positioning can lead to unexpected results, it is still possible to set the desired coordinate system at any point during run-time. The responsibility for choosing a system before starting the scene setup is left to the developer.

Since changing the coordinate system of a project during development would lead to quite unexpected behavior, this constraint appears acceptable. We will be using the OpenGL standard as the default coordinate system in our library if no other system is set explicitly.

Although such relative axes are defined in all graphics engines, they are not an integral part of the API. The orientation serves a more technical purpose: When the scene is viewed from a certain point, every other object is positioned in a coordinate system that has its origin at that point. All objects left of the viewer have a negative X value in OpenGL or Direct3D9 at this point. The mapping of named axes to the relative directions are only performed, as the whole scene is transformed into this coordinate system eventually.

[6]OpenGL and Direct3D9 both use X/Y for left/up but differ in handedness, OpenGL defining negative Z as the forward vector [5], whereas Direct3D9 chooses positive Z [15]. Panda3d, on the other hand, implements a right-handed coordinate system where X, Y and Z correspond to right, forward and upward [13].

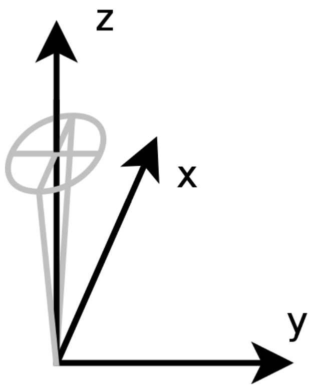


Figure 4.8: A spot light facing "forward" in the Direct3D9 coordinate system, assuming its front-side is the direction the light is emitted to.

The orientation is still used as a default alignment of new objects but a "rotation to the left" does not exist in any of the reference APIs[7], rendering the original definition less useful than it could be. Whatever the orientation of the coordinate system is, one must fall back to the usage of the named coordinate axes X, Y and Z during application development.

We will introduce relative orientation into the API to make stronger use of the already-present orientation capabilities of humans in the real world. If the Direct3D9 alignment of the coordinate system was chosen, a rotation around the Y axis can be expressed as a rotation to the right: `rotateRight(Degree(90));`.

A problem with an approach using this model is the ambiguity of the reference position. A very good example one is immediately confronted with when getting acquainted with a graphics library API is the rotation of a camera. We will start to explore this issue by defining three distinct axes along which one could perform rotations in our relative coordinate system. These axes reach

- from left to right,
- from down to up and
- from backward to forward.

The problem with cameras is that the human brain is not used to process tilted images. A camera that was rotated along the backward-forward axis tends to look odd, as demonstrated by Figure 4.9. This means that the rotational axes for the default use of cameras can be reduced to two:

- from left to right (for looking up/down) and
- down to up (for looking to the sides).

[7]Although Ogre3d provides equivalent methods called `yaw()`, `pitch()` and `roll()`, the wording is intentionally chosen *not* to resemble orientation relative to the current position.

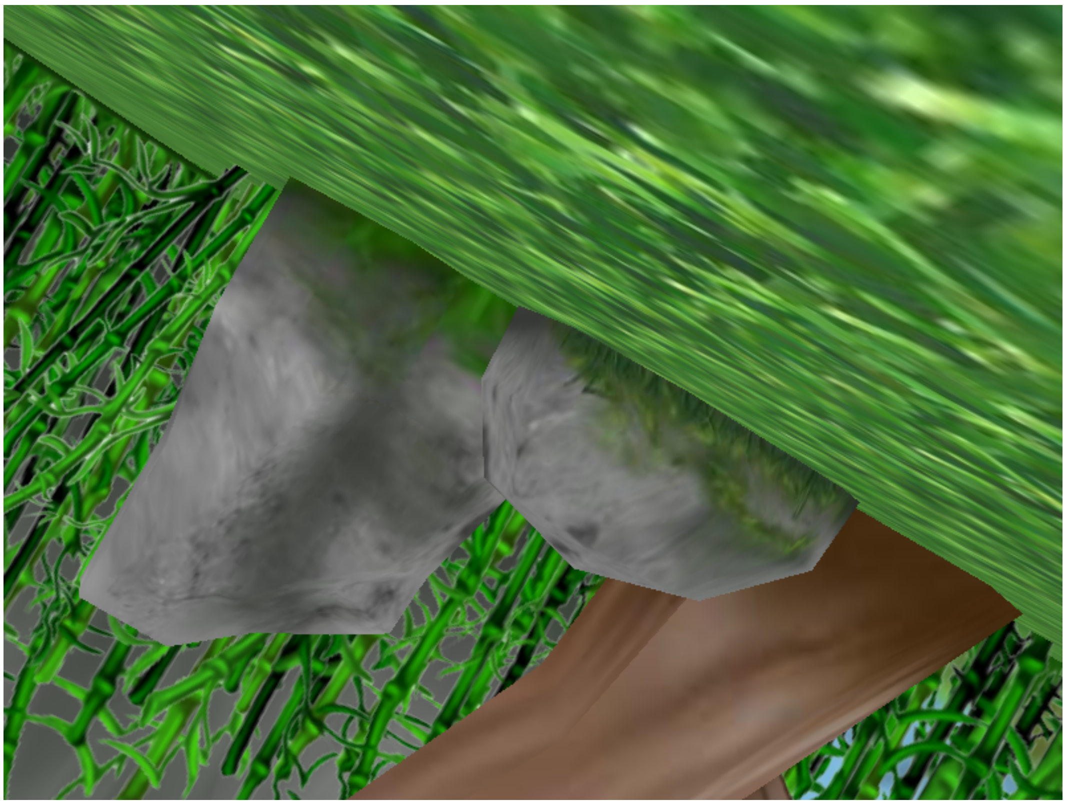


Figure 4.9: A tilted camera

The next issue arises when these two axes are used consecutively. A camera that was rotated to look downward – a rotation along the left-right axis – a rotation along the down-up axis – with the intention of looking sideways – introduces an amount of tilt. This phenomenon is a result of the rotation along a local axis – the tilt of the object has not changed from the viewpoint of the camera object itself, but the tilt of the current orientation of the camera has – from the viewpoint of an unrotated scene node, as seen in Figure 4.10.

The available rotation axes to a camera are further constrained to

- the *local* axis traversing from left to right and
- the *global* axis pointing upwards.

This example demonstrates the main issue with rotations. Every rotation consists of two distinct components:

1. an axis/angle pair that defines the rotation and
2. the coordinate system the rotation is to be performed in.

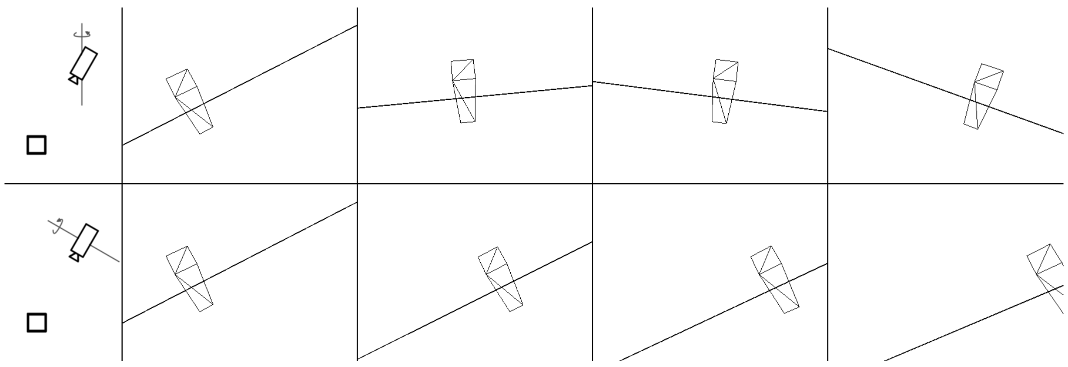


Figure 4.10: Rotating a camera looking down on a cube around the global coordinate system (upper frames) and its local coordinate system (lower frames). The lower images demonstrate that the tilt of the camera has not changed within its local coordinate system: the line on the floor is always parallel to that in other frames.

There are two approaches in our reference engines to the ambiguity arising due to the non-commutative nature of quaternion multiplication during rotations. Prior to these solutions, we must take note that all reference engines use quaternions for the orientation within the scene. The following is an example in Ogre3d:

```
Quaternion rotation1;
Quaternion rotation2;
rotation1.FromAngleAxis(Degree(90), Vector3::UNIT_Y)
rotation2.FromAngleAxis(Degree(90), Vector3::UNIT_X)
sceneNode->rotate(rotation1 * rotation2);
```

The quaternions are used differently by our reference engines, though:

- Ogre3d's `SceneNode` has a method `rotate()` that accepts a quaternion as first, and the `TransformSpace` as second parameter. The `TransformSpace` is an enum consisting of the values
  - `TS_LOCAL` – rotation within the current orientation, this is the default value of this parameter
  - `TS_PARENT` – rotation within the orientation of the parent scene node
  - `TS_WORLD` – rotation from the viewpoint of an unrotated scene node
- The other libraries do not provide any means of rotating rotated nodes at all. The only option is to *re-set* the orientation of a scene node. The developer is responsible for the creation of the desired quaternion for this purpose.

Although Ogre3d uses three predefined context spaces, the rotation is not limited to those three, so we will approach the issue from a different angle. Instead of limiting the contexts to these values, we will declare that a rotation requires a *context node* the operation is performed in. In the camera example above, this is either the parent node (casually referred to as “local” value above), or the root node of the scene graph (the “global” value).

Every node should actually be viewed as a coordinate system that is embedded into another coordinate system[8]. In other words: every child node creates a sub-space within its parent node. This means that the global result of a rotation will change depending on the space the query is performed in.

```
sceneNode->rotate(quaternion, context);
```

This choice brings up an ambiguity: it is not clear, whether the `context` implies an *orientation* or a *location*. The above code snippet could be interpreted as either of:

- “Perform the rotation as if the `context` node was looking at `sceneNode` from its current *position*.”
- “Perform the rotation as if the `context` node was re-positioned to look at `sceneNode` without changing its *orientation*.”

We have accepted this flaw and kept this solution, since the behavior during re-orientation is application-specific – a camera would probably be rotated differently than an asteroid floating in space without gravity. Re-positioning a node leaves much less room for interpretation: the only remaining variable is the distance to the object, which is irrelevant for this purpose.

The implementation of the concepts described here can be found in Section 5.1.

4.5 Scene Construction

Scene graphs are opaque in all of the reference engines, requiring the designer of a scene to use these constructs explicitly when constructing a scene. Panda3d declares that a `Model` is merely a special `SceneNode` that can handle rotating, movement and scaling, while Ogre3d considers `Model`s to be `Object`s and moves them outside the scene graph hierarchy. OpenSceneGraph takes an even stricter approach and defines that such operations can only be performed on nodes dedicated to these tasks.

Creating a port of the scene depicted in Figure 2.1 in Ogre3d could look as follows:

```
// create scene nodes
tableRootNode = roomNode->createChildSceneNode("TableRootNode")
tableNode = tableRootNode->createChildSceneNode("TableNode")
cutleryNode = tableNode->createChildSceneNode("CutleryNode")
forkNode = cutleryNode->createChildSceneNode("ForkNode")
knifeNode = cutleryNode->createChildSceneNode("KnifeNode")
// create models
tableModel = sceneManager->createEntity("TableModel", "table.mesh");
```

[8]Except for the root node, which does not have a parent

```
forkModel = sceneManager->createEntity("ForkModel", "fork.mesh");
knifeModel = sceneManager->createEntity("knifeModel", "knife.mesh");
tableNode->attachObject(tableModel);
forkNode->attachObject(forkModel);
knifeNode->attachObject(knifeModel);
// position objects
cutleryNode->setPosition(1, 1, 10);
forkNode->setPosition(0, 0, 0);
knifeNode->setPosition(1, 0, 0);
```

The choices in Ogre3d and OpenSceneGraph lead to the creation of additional objects during the construction of a scene, though. Every model needs its own scene node to allow us to position the object within its parent node. Panda3d's approach makes this task a bit easier:

```
// create scene nodes
tableRootNode = window->get_render().attach_new_node("TableRoot");
cutleryNode = tableModel.attach_new_node("cutlery");
// create models
tableModel = window->load_model(tableRootNode, "table");
forkModel = window->load_model(cutleryNode, "fork");
knifeModel = window->load_model(cutleryNode, "knife");
// position models
cutleryNode.set_pos(1, 1, 10);
forkNode.set_pos(0, 0, 0);
knifeNode.set_pos(1, 0, 0);
```

After positioning all objects with either of the code snippets above, the table and all other objects can be moved, rotated or scaled at once using the `tableRootNode` object, which provides the appropriate methods for these operations (`setPosition`, `setRotation`, `setScale`).

We will adopt Panda3d's model for the implementation of the scene graph, where every `Model` is a `SceneNode` itself, providing all the functionality required for positioning of the object. This idea is not limited to `Model`s, every tangible object can have its own `SceneNode` sub-class that provides additional methods for the modification of the object itself. This approach naturally reduces the number of steps required for the implementation of a scene.

A drawback of this model became clear quite early during the design process: It does not differentiate objects from their incarnations within the scene. This distinction is required to embed the exact same object multiple times into a single scene. As `Model`s have no properties of their own in our library, the problem became visible during the implementation of the `SpotLightNode`.

When placing a modifiable object into the scene, we felt the need for a distinction of the object being placed and its position parameters in the scene. Some properties of a single `SpotLight` could need to be updated, regardless of their position in the scene. There needs to be a `SpotLight` and a `SpotLightNode`. We will introduce two distinct classes for this purpose:

- the `SpotLightDefinition` that describes the object and
- a `SpotLightNode` that contains information about placement of the object within the scene.

There can only be one `Definition` of an object, but multiple `Node`s integrating that object into the scene. We will be using the spotlight color as a modifiable property during the rest of this design process, although conceptually, there is no difference between any of the object types[9] to be integrated into a scene graph. The API treats them all in the exact same way.

The `SpotLightNode` is a sub-class of `SceneNode` and embeds a `SpotLightDefinition` into the scene graph. To unify the API between the two classes – to make them feel alike – we will additionally introduce an interface called `SpotLight`. This design is visualized in Figure 4.11.

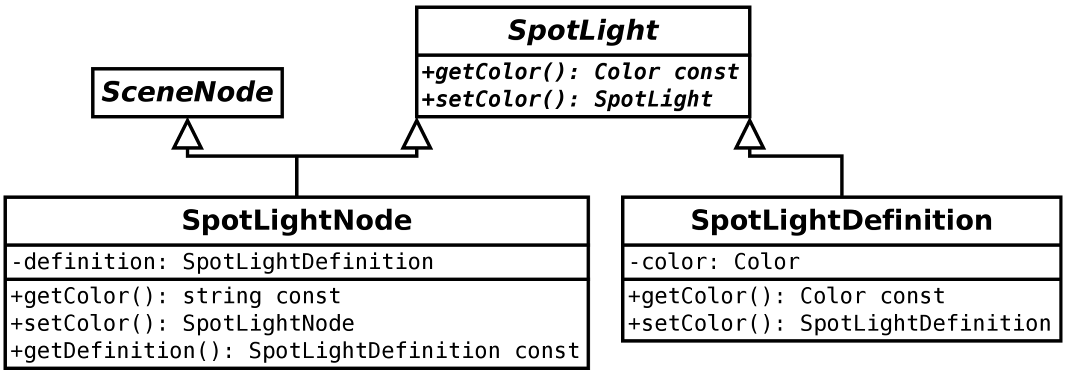

SpotLight
+getColor(): Color const
+setColor(): SpotLight
SceneNode
SpotLightNode
-definition: SpotLightDefinition
+getColor(): string const
+setColor(): SpotLightNode
+getDefinition(): SpotLightDefinition const
SpotLightDefinition
-color: Color
+getColor(): Color const
+setColor(): SpotLightDefinition

Figure 4.11: Interdependencies of the three classes required to define and embed a SpotLight into the scene graph.

- The `Definition` defines a re-usable component, such as a single light source in an array of spot lights for the lighting of a theater stage.
- The `Node` provides the exact same methods for manipulating the underlying template.
- The consequence of this interface is that any operation on the `Node` will alter the appearance of all other nodes in the scene.

We can even conceal this distinction between the `Definition` of an object and its integration into the scene via its `Node` class by creating a factory method in the interface. A `SpotLight` is created and embedded in a single step through a call to this factory:

```
SpotLight* spotLight = SpotLight::create();
```

[9]models, light sources and cameras

The `spotLight` created in this example is of type `SpotLightNode` and it contains a pointer to a newly created `SpotLightDefinition`. Creating an array of lights that have the same behavior is accomplished using the factory in `SpotLightNode`:

```
std::vector<SpotLight*> lights;
lights.push_back(SpotLight::create());
for (int i = 0; i < numLights; i++)
{
    lights.push_back(
        SpotLightNode::create(lights[0]->getDefinition())
    );
}
```

With this design, the scene graph has become quite transparent during scene construction:

- Every created object is immediately attached to the scene graph and
- objects can be attached to each other to declare a dependency of their positional properties.

There is one scenario left, where attaching objects to each other is not enough to declare a dependency between them. This is the case if the objects involved do not have an intuitive hierarchy between them. In our dinner table example, the association between a fork and a knife is such a case. It is not intuitive to declare either of the objects as the one dictating the position of the other, whereas this approach works in the association between a table and a plate: If we move the table around, we want the plate to keep its position relative to the table, but it is hard to say whether the movement of a fork should affect the knife or vice versa.

The API so far would require us to create a scene node to provide a common handle for all the cutlery:

```
cutlery = SceneNode::create();
knife = Model::create("Knife");
fork = Model::create("Fork");
knife->attachTo(cutlery);
fork->attachTo(cutlery);
```

This solution “leaks” the existence of a scene graph. This is not a problem per se, but since this is the only scenario in which the scene graph becomes visible in our API, we will try to address this leakage to eliminate yet another new concept in an API providing an introductory portal into an unknown domain.

We will instead use another mental model that is already present in the user interfaces of many applications that need to arrange objects in 2d or 3d space: object groups. This concept is present in many 3d modeling tools[10], as well as in applications for 2d design[11].

[10]Some examples of well-established modeling tools are *3d Studio Max*, *Blender* or *Maya*

[11]Applications for creating diagrams particularly make use of it, like *dia* or *OmniGraffle*.

An object group has the exact same aim as a node in a scene graph - to introduce a hierarchy into complex object structures. This idea could be implemented in two ways.

1. We could either create a **`typedef`** for the `SceneNode` class, or
2. declare a sub-class thereof.

We have chosen to create a separate sub-class in order to separate it from the `SceneNode` class. A **`typedef`** would remove the black-box status of the `SceneNode`. With this last change, the scene graph data structure cannot be found in the API unless one explicitly seeks it out.

4.6 Scene Modification

When modifying the structure of a scene graph, the objects within the scene are modified implicitly. If a scene node is re-parented to another node, the properties of the child node change. If we consider the example in Figure 4.12, the `getLocation()` method of the `Child` node will return a different value after the re-parenting.

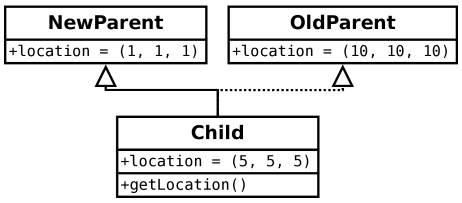


Figure 4.12: Example scene node re-parenting.

An object that has a `location` value of (5|5|5) is positioned with that value as offset from the origin of its parent, no matter where the parent is positioned within the scene. The same is true for other properties of a scene node: rotating the `TableNode` rotates all objects, updating the scale value propagates as well.

The API needs to provide the means of keeping its property on a global scale. The object should be able to keep its location from the viewpoint of an observing camera, otherwise it will jump to another position in-between two frames. As none of our reference engines supports this operation, we will have to find a new solution.

The adjustment of the object's properties is trivial from a mathematical stand point, so we could just allow the developer to pass the properties he/she wants to keep on a global scale as a parameter:

```
child->attachTo(newParent, POSITION | ORIENTATION);
```

But we had a similar problem during the design of relative rotations in Section 4.4. The choice at that point was to support passing arbitrary nodes as a point of reference, not limiting the developer to a set of predefined values. We will try to provide the same flexibility for this operation.

Updating the property of a scene node while retaining its value relative to another node can be performed with these two steps:

- The property value is queried from the viewpoint of the *context node* before the object is detached from its `OldParent`.
- The value is recalculated to match the previously queried value in the same context after the parent was changed.

Finding a solution to these two operations is the foundation of the solution to the initial problem. We had already addressed the second operation in our previous discussion on relative orientation. The other operation could be performed on the same basis, leaving us with the following code sample:

```
location = child->getLocation(context);
child->attachTo(newParent);
child->setLocation(location, context);
```

We can further create a facade that updates all properties of the child node to match its previous values from the viewpoint of a context node:

```
child->attachTo(newParent, context);
```

Apart from the issues during re-arrangement of the scene graph, we will tackle another aspect of the API. We had discussed the possibility to register `Task`s to be performed during the render cycle. We found that one usage scenario when getting familiar with a graphics API was the movement of objects. We were creating a separate class for updating the location of an object along a vector in each application for testing a graphics library.

We decided to implement a dedicated `Task` for this use case, as this is the most elementary form of scene modification over time. But instead of limiting the class to this single operation (movement in a single direction), we will be implementing all basic operations – movement, rotation and scaling.

These thoughts have led us to the design of the `SceneNodeModificationTask`, which has the exact same interface as a `SceneNode`. A basic class outline is shown by Figure 4.13. An object of this type performs the operation repeatedly on an object as long as it is present in the main task group.

To configure the movement of an object along a vector, one needs to provide the vector at the length of the desired distance the object shall travel *per second*. Implementing an object moving at a speed of 100 along the vector (1|1|1) can be accomplished by the following code snippet:

```
task = ModelModificationTask::create();
task->moveForward(Vector3(1, 1, 1).resize(100));
task->register(model);
MainTaskGroup::get()->add(task);
```

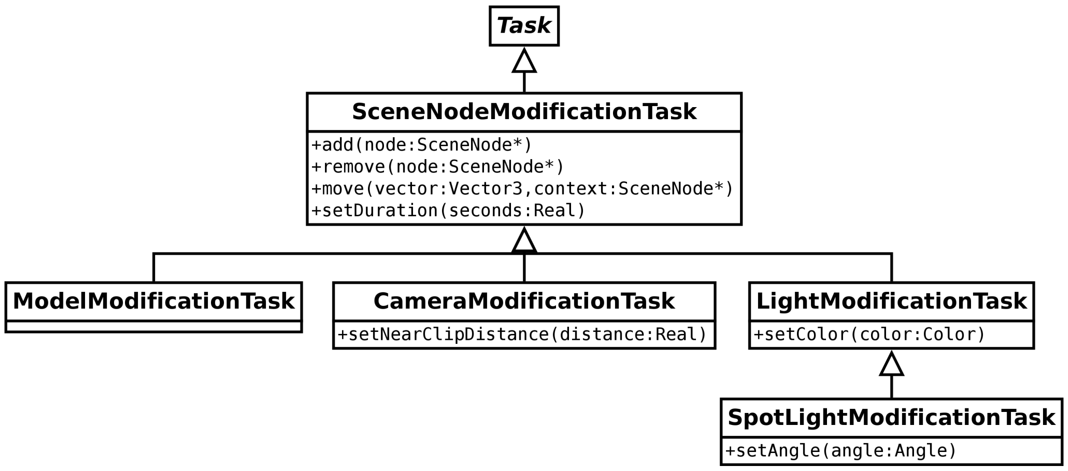


Figure 4.13: Outline of the `SceneNodeModificationTask` classes. Note that the classes lack several methods that would unnecessarily clutter the diagram.

This `Task` object will now repeatedly call the `moveForward()` method on the `model` at each loop cycle. The parameter vector will be re-sized during each call to have the correct length in relation to the time passed since its last call. The same procedure can be performed with rotations as well as scalars.

The same concept can be applied to other properties of other objects. Dimming a spot light over the course of two seconds can be achieved with the following code:

```
task = LightModificationTask::create();
task->setColor(Color::BLACK);
task->setDuration(2);
task->register(spotLight);
MainTaskGroup::get()->add(task);
```

The implementation of these concepts are further described in Section 5.1.

CHAPTER 5

Implementation

After outlining the big picture in the previous Chapter, we will look at the implementation details of some components. We have noticed that the majority of the implementation ideas revolve around the scene graph.

5.1 Scene Graph

The basic design of the base class for all specialized scene nodes is presented in Figure 5.1. Note that there are many more methods in the actual implementation that were only added for convenience. We will look at the implementation of all methods in this graph, dividing them into two categories:

- Re-parenting
- Property Queries and Modifications

We will additionally look into the implementation of some convenience methods and patterns that try to further simplify the API.

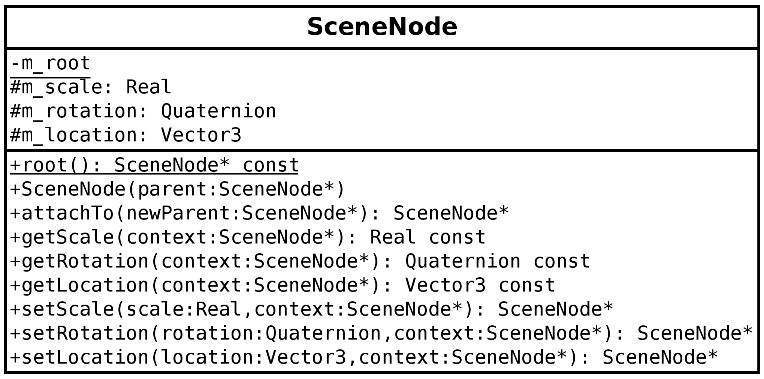
Re-parenting

When arbitrary re-parenting of scene nodes is supported in an API, there is the possibility to create circular dependencies:

```
knifeNode->attachTo(forkNode);
forkNode->attachTo(knifeNode);
```

We have three solutions to this issue: Either

- the API is designed in a way that makes it impossible to create any circular dependencies,



```
SceneNode
-m_root
#m_scale: Real
#m_rotation: Quaternion
#m_location: Vector3
+root(): SceneNode* const
+SceneNode(parent:SceneNode*)
+attachTo(newParent:SceneNode*): SceneNode*
+getScale(context:SceneNode*): Real const
+getRotation(context:SceneNode*): Quaternion const
+getLocation(context:SceneNode*): Vector3 const
+setScale(scale:Real,context:SceneNode*): SceneNode*
+setRotation(rotation:Quaternion,context:SceneNode*): SceneNode*
+setLocation(location:Vector3,context:SceneNode*): SceneNode*
```

Figure 5.1: Basic layout of the `SceneNode` class.

- the issue is ignored altogether as such a circular construct is no longer part of the scene graph and does not have any impact on the visual output, or
- the library checks for them at certain points during the execution.

The only available solution eliminating the problem altogether is the prohibition of re-parenting scene nodes. It is not possible to generate circular dependencies if an existing parent node is responsible for creating new nodes and those parents cannot be re-attached to other nodes. We will discard this very effective solution, as it renders the scene graph API very inflexible.

The next option is to ignore such constructs. As every node has exactly one parent, a circle implies that it does not have a reference to the root scene node. Since the scene only contains nodes that are somehow attached to the root node, these nodes won't be displayed at all. This choice could lead to bugs that are quite hard to find for the same reason.

The remaining option involves verifying the tree structure either when the scene graph changes or when the parent of a scene node is queried. Since the parent node is expected to change less often than it is queried, the check for circular references is performed immediately after an object is attached to another one. This choice additionally provides much more concise error messages and better supports debugging of such an error case.

Property Query and Modification

Scale

The scale of a node in the global context is the product of all scale factors in all parent nodes: $s_g = s_{p_0} \cdot s_{p_1} \cdot \ldots \cdot s_{p_n} \cdot s_n$. Evaluating the scale of a scene node from the viewpoint of another node, we just need to find the quotient of the global scale values of the viewpoint and the queried node:

```
Real SceneNode::getScale(SceneNode* context) {
    if (context == parent) {
        return m_scale;
    }
    if (context == root()) {
        return parent->getScale(root()) * m_scale;
    }
    return getScale(root()) / context->getScale(root());
}
```

Although this implementation would be sufficient, we have added another dedicated if-block for handling the case where `context == this`, as the return value is always 1 in this case. Space never seems distorted from ones own viewpoint.

After the implementation of the query method, the implementation of the setter becomes trivial:

```
SceneNode* SceneNode::setScale(Real s, SceneNode* context) {
    m_scale *= s / getScale(context);
    return this;
}
```

Rotation

Rotations are stored as a rotation within the coordinate system of the parent node. To evaluate the rotation of a scene node in the global context, we will need to multiply the rotations of all parent nodes. The return value of `node->getRotation(RootSceneNode::get())` is computed by $r_g = r_{p_0} \cdot r_{p_1} \cdot \ldots \cdot r_{p_n} \cdot r_n$, where r_n denotes the rotation quaternion of the current node and r_{p_0} through r_{p_n} are the rotation quaternions of the parent nodes.

To evaluate the rotation of a node in the coordinate system of another node, we can take the rotations of both nodes in the global coordinate system and compute the necessary quaternion that would change the rotation of the viewpoint to that of the target. If r_g and R_g are the global rotation quaternions of the target and context nodes respectively, the rotation from R_g to r_g can be computed using $\Delta r = R_g^{-1} \cdot r_g$.

The resulting quaternion Δr can be used to rotate R_g to r_g, as $R_g \cdot \Delta r = R_g \cdot R_g^{-1} \cdot r_g = r_g$. Surprisingly the steps required for this computation do not differ from the evaluation of the scale. This defines the implementation of `getRotation()` method very similar to `getScale()`:

```
Quaternion SceneNode::getRotation(SceneNode* context) {
    if (context == parent) {
        return m_rotation;
    }
    if (context == root()) {
        return parent->getRotation(root()) * m_rotation;
    }
    return context->getRotation(root()).inverse() *
            getRotation(root());
}
```

The method for updating the rotation is analogous to `setScale` as well:

```
SceneNode* SceneNode::setRotation(Quaternion r, SceneNode* context) {
    Quaternion perceivedRot = getRotation(context);
    m_rotation *= perceivedRot.inverse() * r * perceivedRot;
    return this;
}
```

Location

Finding the location of an object is a bit more complex, as the calculation involves the other two properties. The position in global space is defined by

$$l_g = l_{p_0} + (r_{p_0} \cdot l_{p_1} \cdot s_{p_0}) + \ldots + (r_{p_{n-1}} \cdot l_{p_n} \cdot s_{p_{n-1}}) + l_n$$

This is necessary as the *rotation* and *scale* properties do not effect the node they have been defined in but solely effect the sub-space they are creating.

This makes the calculation of a relative location to another more complex as well. After finding the global positions of both nodes (the queried node and the context node), we can create a vector pointing from one point to the other. We will need to adjust that vector using the rotation and scale of the own values of the context node:

$$\Delta l = R_n \cdot (l_g - L_g) \cdot S_n$$

After that we can use Δl to position an object in the space of the context node at the exact same position as the initially queried node. The implementation of this method explicitly requires the handling of the context node being the parent node this time:

```
Vector3 SceneNode::getLocation(SceneNode* context) {
    if (context == parent) {
        return m_location;
    }
    if (context == root()) {
        return parent->getLocation(root()) +
               parent->m_rotation * m_location * parent->m_scale;
    }
    return getLocation(root()) - context->getLocation(root());
}
```

Setting the location from an external viewpoint is as simple as the setter of the other two properties:

```
SceneNode* SceneNode::setLocation(Vector3 l, SceneNode* context) {
```

```
m_location += l - getLocation(context);
    return this;
}
```

Convenience

Additional Methods

The methods presented at the beginning of this Section are completely sufficient to perform all available operations on `SceneNode`s. Being limited to these few operations would render the API unnecessarily complex. We will try to hide any design details until the developer actively seeks them out. We have implemented a wide variety of additional methods for this purpose:

- Property modification without context, where the context is assumed to be the parent node:
  - setScale(Real scale)
  - setRotation(Quaternion rotation)
  - setLocation(Vector3 location)
- Relative property modification:
  - scale(Real scalar)
  - rotate(Quaternion rotation)
  - move(Vector3 vector)
- Relative movement and rotation, both with and without the additional context parameter:
  - moveLeft(Real amount)
    moveRight(Real amount)
  - moveUp(Real amount)
    moveDown(Real amount)
  - moveForward(Real amount)
    moveBackward(Real amount)
  - rotateLeft(Angle amount)
    rotateRight(Angle amount)
    turnLeft(Angle amount)
    turnRight(Angle amount)
  - rotateUp(Angle amount)
    rotateDown(Angle amount)
    turnUp(Angle amount)
    turnDown(Angle amount)
  - tiltLeft(Angle amount)
    tiltRight(Angle amount)

The definition of these methods have bloated the API on one hand, but provided the perfect method for many use cases. We have noticed during testing that we were not using quaternions at all. We were able to create our test scenes just with relative modification of our objects.

Fluent Interfaces

When embedding an object into the scene graph, we need to adjust several properties of the object. Usually this involves multiple distinct calls to a newly created scene node. A pattern that we immediately thought of with this insight was the fluent interface [11]. Fluent interfaces return the object that is being operated on to allow us to chain multiple methods. The above example would be reduced to just four lines of code with the implementation of this idea:

```
table = Model::create("Table");
cutlery = SceneNode::create()
    ->setPosition(1, 1, 10)
    ->attachTo(table);
knife = Model::create("Knife")
    ->setPosition(1, 0, 0);
    ->attachTo(cutlery);
fork = Model::create("Fork")
    ->attachTo(cutlery);
```

Unfortunately, we had chosen to sub-class the `SceneNode` for each object type that can be attached to the scene graph. This means that any operation on the `SceneNode` API would return a `SceneNode` object and no longer its sub-class. But a fluent API returning the covariant type would be much more intuitive to avoid the following scenario:

```
light = SpotLight::create()
    // the next call would return a SceneNode object
    // which does not provide the setColor() method:
    ->setPosition(1, 1, 1)
    // compilation error on the next line:
    ->setColor(Color::RED);
```

The simplest solution would be to re-implement all methods of the `SceneNode` in the `SpotLightNode` class to return the type of the implementing class. Instead of performing this implementation in each sub-class of `SceneNode`, we will use the *curiously recurring template pattern* (CRTP) to accomplish this task. [21] presents this pattern based on work in [2] and its implementation in our library is presented in Figure 5.2

The API now returns the type of the object an operation was called on, allowing us to use all methods of a `SpotLightNode` in succession, regardless of the implementation of the methods within the class hierarchy.

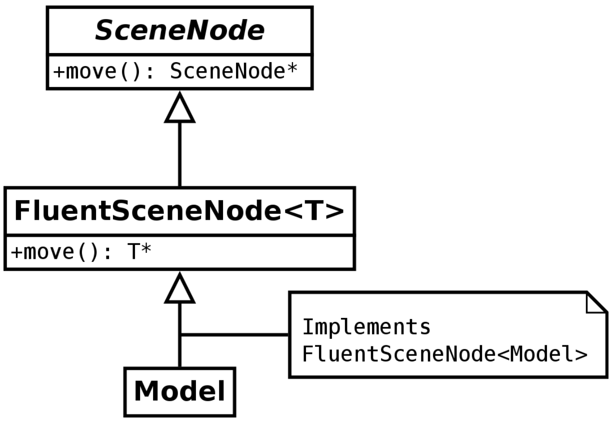


Figure 5.2: The architecture of the fluent interface. The template class' method `FluentSceneNode<T>::move()` executes `SceneNode::move()` and returns **`this`**, casting it to a `T` pointer.

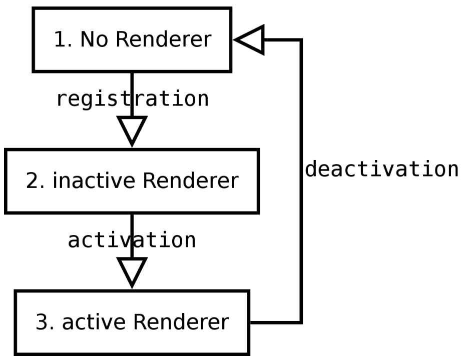


Figure 5.3: All states and some possible state changes regarding the external renderer.

5.2 Renderer Communication

The external renderer is implemented as a replaceable component of the library. Because of this approach, the communication between this component and the rest of the library needs to be divided into several states, depending on the presence of the renderer and whether it is active or not. The flowchart in figure 5.3 shows the state changes between

1. the initial state of the library without any registered renderers,
2. the operation with an inactive renderer and
3. the state an application usually runs in – where the output of the scene is being rendered.

The flow of information between PURGE and its rendering component was defined as follows:

- No registered renderer: All operations on objects are performed within PURGE, any parameter changes are stored in the main library. This state guarantees that the API is capable of processing all commands independently of any implementing renderers.
- As soon as a renderer has been defined, it is registered as an available component in PURGE, but there is still no flow of information between the two layers.
- The activation of the renderer causes the first flow of data. The renderer itself will first call its own initialization routines, acquire the resources it needs, and will then receive the data forming the current scene. When this process is finished, the renderer will be called regularly by the main loop to render the scene.
- Once the renderer is de-activated, the renderer is expected to free all resources in order to allow another renderer to be activated – which will possibly require some resources this renderer was holding.

As outlined in the description of the state changes, there are two instants at run-time that require passing scene data from PURGE to the renderer:

1. The activation of the renderer and
2. the rendering process, which will periodically need access to the scene data in order to render it.

The first instance requires the transmission of all data that was generated in the absence of the renderer. The second state, however, requires merely the transmission of changes – re-evaluating every existing object in the renderer would have an easily avoidable performance impact.

A possible implementation could consist of a list of operations that were performed on the scene objects, which can be flushed to the renderer in both cases. This approach supports the equal treatment of both scenarios: In the first case, all operations are transmitted to the renderer, in the second case only those that have been triggered since the last transmission.

This solution requires the tracking of any changes to the scene, which will be flushed to the renderer before the actual rendering is performed. Unfortunately, this necessity requires further optimizations to eliminate the increasing memory consumption throughout the lifetime of the application.

The next approach is a variation on the first. Several distinct events have been defined that can be consumed by the renderer. At each cycle, the following operations are accumulated:

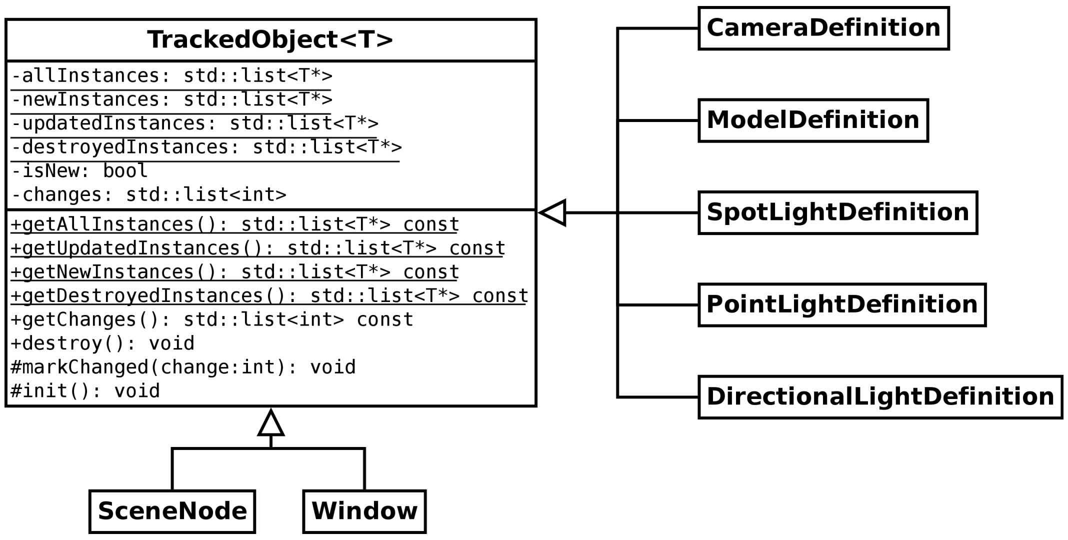
- newly created objects,
- updated objects and
- destroyed objects.

The renderer is expected to process these lists of events to implement the specified scenario. PURGE, on the other hand, is responsible for the proper accounting: Each object is present in at most one list, which ensures that

- objects that were destroyed in the same cycle are not present in any of these lists.
- newly created objects are not present in the list of updated objects, even if any modifying operations were performed after their creation.
- destroyed objects do not appear in any other registers.
- objects that were not modified at all during the current cycle are not present anywhere.

Updated objects further keep a list of properties that were updated during the cycle. The new values of these properties can be queried by the renderer as needed. Changing the color of a `SpotLight` thus leads to the following operations within the render cycle:

1. The object is marked as changed – it is added to the list of updated `SpotLight` objects.
2. The value `SpotLight::CHANGE_COLOR` is pushed onto the vector of changes of the same object.
3. When the renderer is instructed to perform the rendering, all updated SpotLight objects are collected via `SpotLight::getUpdatedInstances()`, the changes to the object are retrieved and the color value of the renderer's internal `Light` object is replaced by the new value retrieved by `purgeSpotLight->getColor()`.



TrackedObject<T>
-allInstances: std::list<T*>
-newInstances: std::list<T*>
-updatedInstances: std::list<T*>
-destroyedInstances: std::list<T*>
-isNew: bool
-changes: std::list<int>
+getAllInstances(): std::list<T*> const
+getUpdatedInstances(): std::list<T*> const
+getNewInstances(): std::list<T*> const
+getDestroyedInstances(): std::list<T*> const
+getChanges(): std::list<int> const
+destroy(): void
#markChanged(change:int): void
#init(): void
CameraDefinition
ModelDefinition
SpotLightDefinition
PointLightDefinition
DirectionalLightDefinition
SceneNode
Window

Figure 5.4: Outline of the `TrackedObject` class.

Figure 5.4 shows an outline of the base class for all objects that need to be considered during the rendering process. This template class using the curiously recurring template pattern provides the necessary query functions for the `Renderer`. There are some considerations for the usage of such a class for all sub-classes:

- The constructors of inheriting classes must call the `init()` method of the parent for the registration of the object in the correct lists.

- Changes to properties must be registered at the base class via a call to `markChanged()`, passing a constant value describing the property that was updated.
- Sub-classes must not be destroyed via their regular destructors. An object that is to be removed from the application needs to be marked with a call to `destroy()`, as the destroyed object must be communicated with the `Renderer`.

With such an interface to the objects in the graphics engine, the `Renderer` is capable of pulling just the information it needs to update its own objects. The exact management of the `Instances` arrays is described in Section 5.4.

5.3 OgreRenderer

The implementation of the `Renderer` class using the Ogre3d library posed a few interesting problems.

Initialization

Resource Locations

PURGE assumes that there is a pool of `Models` available for usage that can be addressed with a unique string identifier each. Fortunately, Ogre3d takes the same approach: During the initialization of the engine, one has to define a list of resource locations that contain such named objects. This means that the initialization of the `OgreRenderer` itself must provide access to the functionality already present in Ogre3d.

As this is the only parameter that needs to be specified at the start-up of the engine[1], we have decided to accept a resource location as a parameter to the factory.

Ogre3d supports two different resource location classes containing resources – both deriving from the `Archive` interface: the `FilesystemArchive` and the `ZipArchive`, one providing access to files in a directory, the other granting access to files in zip-archive. As our library is aimed at rapid prototyping, we have chosen to support a single directory name as a resource location. Additional locations can be added using the API of Ogre3d itself, if required.

Delayed Initialization

Another issue arose during the implementation of the renderer on Linux. The application was crashing if an Ogre::Camera was created an Ogre::Window. A similar crash occurred when the resource locations were registered before a Window was established. Both issues turned out to be a constraint imposed by OpenGL: Several operations in OpenGL require a window as a context.

The solution was to delay the initialization of the complete `OgreRenderer` until a `PURGE ::Window` was defined. This approach has no side effects, as the rendered output on the window

[1]All other parameters can operate with default values, but the location of the resources required by the application are taken to be of great interest to the developer using our API.

is the single duty of the `Renderer`. The method `perform()` would return immediately without performing any operations either way.

Communication

The communication between the two libraries is handled at a central point, the `Transcription` class, outlined in Figure 5.5. This template class uses the curiously recurring template pattern – just like its management counterpart in PURGE, the `TrackedObject`, outlined in Figure 5.4 – and provides two static functions for synchronizing PURGE objects and their corresponding bridge objects:

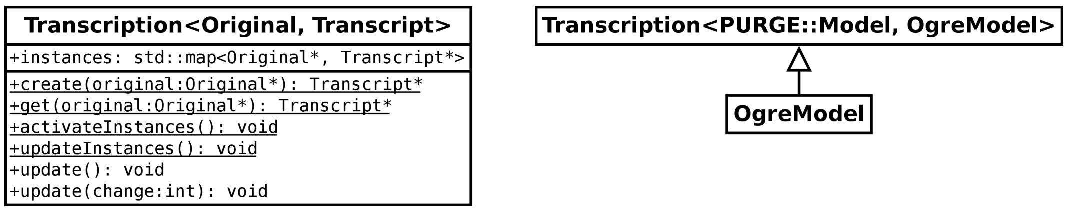

Transcription<Original, Transcript>
+instances: std::map<Original*, Transcript*>
+create(original:Original*): Transcript*
+get(original:Original*): Transcript*
+activateInstances(): void
+updateInstances(): void
+update(): void
+update(change:int): void
Transcription<PURGE::Model, OgreModel>
OgreModel

Figure 5.5: The `Transcription` class manages the communication between PURGE objects and Ogre3d objects.

- `activateInstances()` will read all available objects in PURGE and create matching instances in the `OgreRenderer`. This step is only necessary to activate the renderer.
- `updateInstances()` is called by `OgreRenderer::render()` prior to the rendering. It creates new objects, destroys obsolete objects and carries any modified state from the existing PURGE objects to the bridge objects.

Both procedures possibly create bridge objects. To provide greater flexibility for this purpose, the classes assume a static factory method instead of calling the constructor directly. The `Transcript` template parameter has the option of implementing this factory to perform additional operations after the object was created.

5.4 Main Loop

The implementation of the render loop designed in Section 4.2 is outlined in Figure 5.6. The `Task` class is the base class for all operations to be performed in-between the rendering steps. The `TaskGroup` combines multiple `Task`s into a single `Task`.

The current implementation of `TaskGroup::perform()` calls all registered `Tasks` sequentially. An alternative to this approach would be the implementation of a `ParallelTaskGroup`. This idea was not implemented, though, as this would require the introduction of thread-safety to the application as a whole. This additional option could be pursued at a later time as an optimization technique to the PURGE library.

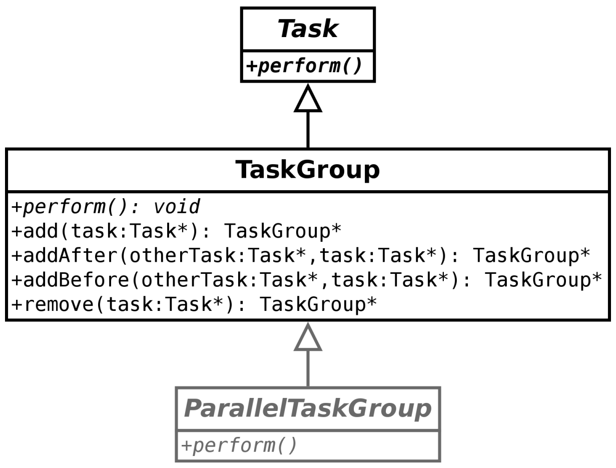

Task
+perform()
TaskGroup
+perform(): void
+add(task:Task*): TaskGroup*
+addAfter(otherTask:Task*,task:Task*): TaskGroup*
+addBefore(otherTask:Task*,task:Task*): TaskGroup*
+remove(task:Task*): TaskGroup*
ParallelTaskGroup
+perform()

Figure 5.6: Outline of the `TaskGroup` class.

RenderTask

PURGE further needs to provide some tasks by itself covering the rendering step. These special tasks are responsible for passing control to any active `GraphicsImplementer` to generate the visual output of the scene. Following the philosophy of the Ogre3d rendering loop, this step is defined by the following `Task` classes:

- RenderingPreparation: This `Task` is an equivalent of Ogre3d's `frameRenderingQueued` event. When this task is finished, all operations that need to be performed on the CPU are finished and the GPU is working in the background.
- RenderingFinished: This `Task` will block until the GPU has finished its pending operations and the next cycle of the render loop can be entered.

This distinction allows one to inject tasks in-between these two steps, achieving the same flexibility found in Ogre3d. An example task queue is outlined in Figure 5.7.

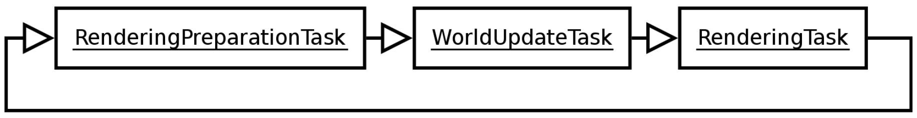


Figure 5.7: A very simple rendering loop.

CHAPTER 6

Evaluation

We have designed several test scenes to test various aspects of the final API. The tests scenarios are used for obtaining source code metrics, as well as for performance comparison between the implementations with and without the PURGE layer.

All source code measurements exclude statements in reference engines that have no equivalence in the simplified PURGE API. Furthermore, many test scenarios include source code from previous tests. To keep the focus on the current test, all statements that were covered in a prior case are ignored.

The measured values are:

- Lines of code: Number of lines in the source code of the application. Each line contains a single statement, empty lines and lines consisting of a single brace are ignored. Some tests further do not include the placement of a camera, which is required to test the number of rendered frames per second (below), thus the statements required for this positioning are ignored, too.
- Function calls: Number of API functions that were used for the implementation.
- Distinct function calls: This value represents the number of different functions that were needed to implement the scenario. It is assumed that a lower value reduces the complexity of the API.
- Average frames per second: Number of frames that were drawn until the end of the test application. The value is the average of 100 runs of the test scenario, rendering for five seconds in each iteration. It is important to note that this value is highly dependent on the bridge implementation in the case of PURGE.
- Peak memory usage: This value is obtained through the `getrusage()` system call on Linux.

6.1 Creating a scene

The first 5 test cases will gradually create an application that will render a space ship at the center of the render window. The finished scene is depicted in figure 6.1

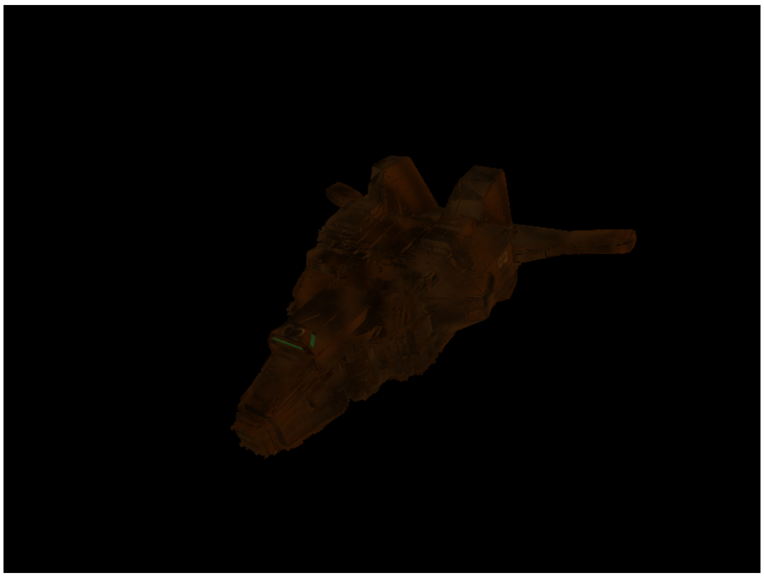


Figure 6.1: Output of TestScene5

TestScene1: Creating an empty render window

The minimal code required to create a window and enter a render loop that will exit whenever the window is closed. This is the most basic scenario for every graphics engine. This test measures the load of the boilerplate code of each graphics library. A performance comparison is not sensible in this case, as PURGE chooses to create a differently sized default render window than Ogre3d.

Table 6.1: Code metrics for TestScene1

	PURGE	Ogre3d	OpenSceneGraph	Panda3d
Lines of Code	2	8	1	4
Function calls	3	9	2	4
Distinct Function calls	3	9	2	4

Table 6.1 shows that OpenSceneGraph needs the fewest lines of code for this task. PURGE has two semantically equivalent statements but requires an additional call for the instantiation of the `Renderer`. Additionally this test provides the first assertion of the initial statement that Ogre3d is quite verbose.

TestScene2: Controlling the render window

In order to create a test scene that allows direct comparison of the frame rates of the same application with and without the PURGE layer, the render window will be adjusted in this test: The render window is centered on the display and created with a predefined size of 800x600 pixels. Panda3d is the only library that did not need the render window object in the initial test case. The results can be seen in Table 6.2.

Table 6.2: Code metrics for TestScene2

	PURGE	Ogre3d	OpenSceneGraph	Panda3d
Lines of Code	1	1	1	3
Function calls	1	1	1	3
Distinct Function calls	1	1	1	3

This test is the first to additionally measure impact of the added layer to the rendering performance (Table 6.3). As the scene is completely empty (no visible objects have been created), this value shows the resource usage for the idle render loop. This already-low value will further decrease as the impact of the render loop itself is negligible in a graphics application.

Table 6.3: Performance metrics for TestScene2

	without PURGE	with PURGE	relative value
Frames per second	8726.45	8713.35	99.85%
Peak memory usage	76709	77170	100.60%

TestScene3: Positioning the camera

The camera is positioned at $(X|X|X)$ and rotated to look at the origin. Panda3d performs all required steps in one statement each:

1. retrieving the camera object from the window,
2. repositioning the camera and
3. setting the new orientation to look at the global origin.

As OpenSceneGraph provides a method similar to GLU's `gluLookAt()`, the re-alignment can be performed in a single step. PURGE instead makes use of its fluent interfaces to reduce the lines of required code and Ogre3d needs to create a dedicated scene node that performs the transformation for the camera.

The repositioning of the camera did not have any impact on the rendering performance, but the tiny gap between the measured memory consumptions closes a bit further as Ogre3d needs to make use of an additional object.

Table 6.4: Code metrics for TestScene3

	PURGE	Ogre3d	OpenSceneGraph	Panda3d
Lines of Code	1	3	2	3
Function calls	3	4	5	3
Distinct Function calls	3	4	5	3

Table 6.5: Performance metrics for TestScene3

	without PURGE	with PURGE	relative value
Frames per second	8727.26	8712.92	99.84%
Peak memory usage	76709	76934	100.29%

TestScene4: Loading an object

The next step involves loading an object into the scene without modifying its position: the object is attached to the root scene node. As attaching the object to the scene graph is implicit in PURGE, the whole task can be performed in a single step. Ogre3d needs to implement the remaining boilerplate code for registering the location of the resource.

Table 6.6: Code metrics for TestScene4

	PURGE	Ogre3d	OpenSceneGraph	Panda3d
Lines of Code	1	4	1	2
Function calls	1	6	2	4
Distinct Function calls	1	6	2	4

The object used for this test consists of approximately 123.000 triangles. We had assumed that the performance difference would decrease in previous tests and this assumption is backed by the performance difference presented in Table 6.7. The memory footprint of the loaded object further reduces the difference in memory consumption.

Table 6.7: Performance metrics for TestScene4

	without PURGE	with PURGE	relative value
Frames per second	661.75	660.92	99.87%
Peak memory usage	145362	145446	100.06%

TestScene5: Positioning after loading

The previously loaded object is now positioned at the coordinates $(-X|-X|-X)$. Ogre3d and OpenSceneGraph need to create a new scene node that accepts this transformation. A model in Panda3d is itself a scene node that can be manipulated directly. The implementation using PURGE makes use of the fluent interface to append a single function call to an existing line.

Table 6.8: Code metrics for TestScene5

	PURGE	Ogre3d	OpenSceneGraph	Panda3d
Lines of Code	0	2	4	1
Function calls	1	3	4	1
Distinct Function calls	1	3	4	1

The additional scene node required by Ogre3d further closes the performance gap between the two implementations, whereas the difference in memory consumption does not change at all.

Table 6.9: Performance metrics for TestScene5

	without PURGE	with PURGE	relative value
Frames per second	670.15	669.96	99.97%
Peak memory usage	145361	145446	100.06%

6.2 Completed Scene

So far we have looked at the complexity of very specific, incremental tasks. To get a better overview on the amount of code required for the whole scene, we will additionally consider the code metrics of the complete application we have created step by step during the previous five test cases. Table 6.10 shows the difference between all APIs.

Table 6.10: Code metrics for the whole TestScene5 implementation

	PURGE	Ogre3d	OpenSceneGraph	Panda3d
Lines of Code	5	19	10	12
Function calls	8	23	14	16
Distinct Function calls	7	21	14	14

6.3 Modification over time

As the previous tests were testing the creation of a single, static scene, we will be looking at some dynamically updated scene graphs in this section.

TestScene6: Movement

This scene implements a straight movement of a previously loaded object from one point to another. The presence of the automatic scene node manipulator simplifies this task tremendously in the application using PURGE. The Panda3d API is satisfied with a single function, the other implementations need a separate class for an ongoing modification of the scene. In either case the implementation drives the developer through a lot more knowledge than necessary for such a simple task.

The metrics just count the amount of effort required to implement this feature into an existing scene. We will only be measuring the statements we needed to write to make the object in TestScene5 move.

	PURGE	Ogre3d	OpenSceneGraph	Panda3d
Lines of Code	1	8	8	5
Function calls	4	11	12	8
Distinct Function calls	4	11	12	8

Table 6.11: Code metrics for TestScene6

The performance difference between the two implementations was rather small again.

Table 6.12: Performance metrics for TestScene6

	without PURGE	with PURGE	relative value
Frames per second	668.50	668.22	99.96%
Peak memory usage	145361	145454	100.06%

TestScene7: Rotation

This test scene implements a 360° rotation of an existing object. As with the previous test we will only measure the additional statements required for the rotation. Not surprisingly, the results are very similar to that of the previous test.

	PURGE	Ogre3d	OpenSceneGraph	Panda3d
Lines of Code	1	10	8	7
Function calls	4	14	11	11
Distinct Function calls	4	14	11	11

Table 6.13: Code metrics for TestScene7

Table 6.14: Performance metrics for TestScene7

	without PURGE	with PURGE	relative value
Frames per second	670.38	670.27	99.98%
Peak memory usage	145365	145458	100.06%

6.4 Scalability tests

The next few tests will probe how the library behaves when the amount of objects increases. As the tests merely add some loops into the previously analyzed code, we will omit the code metrics for the next tests and just look at the performance. To keep the focus further on the CPU-operations, we have performed all tests in this chapter with cubes of size (1|1|1). More complex objects would shift the majority of the execution onto the GPU, which does the exact same operations in both implementations.

TestScene8: Many objects at origin

We have created one thousand cubes at the origin of a scene for this test. The Ogre3d implementation attaches all objects directly to the root scene node, whereas the `Renderer` of PURGE creates a dedicated scene node for each object. The results clearly show that these additional scene nodes have a huge impact on the performance in the Ogre3d rendering process. The additional nodes further result in increased memory consumption.

Table 6.15: Performance metrics for TestScene8

	without PURGE	with PURGE	relative value
Frames per second	872.94	32.84	3.76%
Peak memory usage	64806	68778	106.13%

This is an uncommon use case scene in graphics development, as there are usually very few objects attached to the root scene node. These are the objects that form the immobile environment of the scene – like the plants and rocks provided by Panda3d's official tutorial application, depicted in Figure 4.9. On the other hand, these models can be extremely complex, straining the GPU in another way. To assess these thoughts, we have created the same test scene with a single, complex object, Ogre3d's official ogre head model consisting of 2242 vertices.

Table 6.16: Performance metrics for TestScene8.2

	without PURGE	with PURGE	relative value
Frames per second	625.87	630.57	100.75%
Frames per second (no far-clipping)	632.73	629.06	99.42%
Peak memory usage	68229	68356	100.19%

Interestingly, PURGE achieved a higher frame rate in this test. This unexpected difference was caused by the default values of the camera frustum in PURGE: we have disabled the far end of the frustum, always drawing everything in sight. This will generate much worse results with a more populated scene, but we will keep the implication that such a scene will require manual tweaking of the camera parameters, optimizing the API for the simplest scenes. Re-running the application after adjusting the Ogre3d implementation provided the results in Table 6.16. We have added this newly found optimization to all following tests.

The elaboration on the initial test showed that the PURGE software layer has trouble with multitudes of objects positioned at the origin. The same scene could be created in another way that would eliminate this deficit entirely: by merging all models to be positioned at the origin into a single model.

TestScene9: Many objects, distributed in space

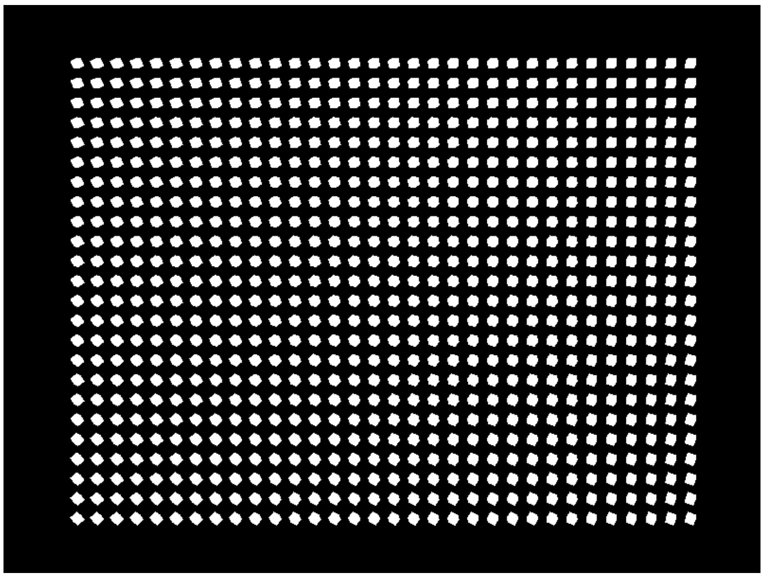


Figure 6.2: Output of TestScene9

We have created a “wall” consisting of $32 * 24 = 768$ cubes facing the camera. The frame rate is again marginally higher with the added software layer, but we couldn’t find the cause for this difference in this scene. It is possible that another choice of a default value effects the performance.

Table 6.17: Performance metrics for TestScene9

	without PURGE	with PURGE	relative value
Frames per second	37.58	37.74	100.43%
Peak memory usage	67708	67702	99.99%

TestScene10: Rotating many objects

The same objects that were created during the previous test were rotated 360° around their up axis throughout the run-time. This test shows that the transmission of changes from PURGE to Ogre3d contains another big performance impact. In contrast to the previous performance deficit in TestScene8, this use case could be much more common and the solution is not as simple.

Table 6.18: Performance metrics for TestScene10

	without PURGE	with PURGE	relative value
Frames per second	73.96	30.38	41.08%
Peak memory usage	79962	83694	104.67%

One possible optimization is sharing of attitude objects. If PURGE has the same handedness of its coordinate system as the underlying graphics engine, it would be possible to write all attitude changes into the quaternion of the underlying engine – or binding the quaternion used by Ogre3d to the object created by PURGE. The same approach could be applied to other properties (like position and scale) as well. This solution breaks the separation of the two graphic engines in favor of a higher frame rate.

Taking the idea of such a sacrifice of separation further, we could implement the newly created API for Ogre3d, rather than a dedicated software layer. This would eliminate the need to transmit any changes from one engine to another, leading to much higher performance.

TestScene11: Many objects, with a higher scene graph level

The layout of the test with distributed objects was repeated, this time attaching the objects in each row to a common parent node in both implementations. The consistent improvement of the frame rate now starts to indicate a positive impact of our layer on scenes consisting of multiple immobile objects, even after re-running the tests.

Table 6.19: Performance metrics for TestScene11

	without PURGE	with PURGE	relative value
Frames per second	40.22	40.33	100.28%
Peak memory usage	67699	67698	100.00%

This effect proved to be the result of the model used in the tests. The results changed as we switched from dynamically created cube models to a pre-calculated cube model loaded from a file. The repeated test summarized in Table 6.20 has slightly lower frame rates than the previous test run, as the pre-calculated cube was larger than the dynamically created one.

Table 6.20: Performance metrics for TestScene11.1

	without PURGE	with PURGE	relative value
Frames per second	34.36	33.92	98.72%
Peak memory usage	67700	67958	100.38%

TestScene12: Rotating the scene nodes

The scene nodes containing each row are rotated around their “up” axis again.

Table 6.21: Performance metrics for TestScene12

	without PURGE	with PURGE	relative value
Frames per second	79.01	74.24	93.96%
Peak memory usage	79966	80495	100.66%

CHAPTER 7

Conclusion

The biggest issue with a learning-by-doing approach in the usage of graphics engines is the huge amount of knowledge their APIs assume. We have managed to create an API that conceals as much of these domain-specific entities as possible and allows the learning developer to choose the order in which he can pick up all these aspects unknown to him.

We found that we could create whole scenes without the explicit need of scene graphs or quaternions, two areas that were very hard to grasp for us while we were familiarizing ourselves with the reference engines. The wide variety of facades and helper classes further simplified our development process. The code metrics of our tests have shown that half the amount of code is sufficient to perform the exact same basic operations as with other graphics engines.

The performance of the library is extremely close to that of the original Ogre3d library in most cases. This gap started growing as we started using more primitive objects in our test scenes. This does not come as a surprise, as the whole library was developed for the communication of very abstract concepts.

7.1 Future Work

We had specifically chosen to cover a very small area of the graphics domain, as the complete domain is huge. Subdividing `Models` to expose their components – like meshes or shaders – is one area that could be interesting. Another approach would be adding interactivity and sound to the library, this could even be sufficient to provide a very simple platform for game development.

On the other hand, the interaction between PURGE and Ogre3d could be optimized. We had already seen a test case where the implementation with an additional layer can provide even better performance than the original implementation. This is actually quite easy to achieve as one could use much more complex computations in the bridging layer than a layman graphics developer could ever think of.

Another approach to the optimization could be a stronger coupling to one specific graphics engine, providing a second API to Ogre3d perhaps. PURGE as an additional software layer could be disbanded to just be a part of the rendering library it was using.

An additional idea emerged during the implementation of the decoupled renderer. The newly created abstraction layer on top of other renderers could be used as a uniform API to the underlying graphics engines. It would be possible to use this API for the comparison of different graphics libraries. This would require more elaborate implementations of the bridging layers, though.

Bibliography

[1] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms for bounding boxes. *J. Graph. Tools*, 5(1):9–22, January 2000.

[2] John J. Barton and Lee R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1994.

[3] Matthias Bauchinger. Designing a modern rendering engine. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 8 2007.

[4] G. Birkhoff and S.M. Lane. *A survey of modern algebra*. Macmillan, 1960.

[5] Samuel R. Buss. *3D Computer Graphics: A Mathematical Introduction with OpenGL*. Cambridge University Press, New York, NY, USA, 2003.

[6] Sharon Rose Clay and Jane Wilhelms. Put: Language-based interactive manipulation of objects. *IEEE Comput. Graph. Appl.*, 16(2):31–39, March 1996.

[7] James Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors, 2006.

[8] David H. Eberly. *3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[9] David H. Eberly. *3D Game Engine Design, Second Edition: A Practical Approach to Real-Time Computer Graphics (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[10] Kenneth Finney. *3D Game Programming All in One*. Premier Press, 2004.

[11] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, January 1995.

[13] Mike Goslin and Mark R. Mine. The panda3d graphics engine. *Computer*, 37:112–114, October 2004.

[14] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 171–178, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[15] W. Jones. *Beginning DirectX 9*. Game Development Series. Thomson/Course Technology, 2004.

[16] F. Kerger. *Ogre 3d 1.7 Beginner's Guide*. Learn by doing : less theory, more results. Packt Publishing, Limited, 2010.

[17] Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6:261–301, December 1974.

[18] Craig Kolb, Don Mitchell, and Pat Hanrahan. A realistic camera model for computer graphics. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 317–324, New York, NY, USA, 1995. ACM.

[19] Bob Kuehne and Paul Martz. *OpenSceneGraph Reference Manual v2.2*. Blue Newt Softwar, LLC, February 2009.

[20] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics, Second Edition*. Charles River Media, Inc., Rockland, MA, USA, 2003.

[21] Stanley B. Lippman, editor. *C++ gems*. SIGS Publications, Inc., New York, NY, USA, 1996.

[22] Paul Martz. *OpenSceneGraph Quick Start Guide A Quick Introduction to the*, 2007.

[23] D. Mathews. *Panda3d 1.6 Game Engine Beginner's Guide*. Packt Publishing, 2011.

[24] R. Mukundan. Quaternions: From classical mechanics to computer graphics, and beyond. In *Proceedings of the 7 th Asian Technology Conference in Mathematics, 2002*, 2002.

[25] Nvidia Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Technical report, Nvidia Corporation, 2009.

[26] Randy Paffenroth, Dana Vrajitoru, Thomas Stone, and John Maddocks. Dataviewer: A scene graph based visualization tool. In *Proceedings of the 20th UK conference on Eurographics*, EGUK '02, pages 147–, Washington, DC, USA, 2002. IEEE Computer Society.

[27] Bob Palais and Richard Palais. Euler's fixed point theorem: The axis of a rotation. *Journal of Fixed Point Theory and Applications*, 2(2):215–220, December 2007.

[28] Jeff Plummer. A flexible and expandable architecture for computer games. Master's thesis, Arizona State University, December 2004.

[29] Andrew Rollings and Dave Morris. *Game Architecture and Design: A New Edition*. New Riders Games, 2003.

[30] Jerome H. Saltzer and M. Frans Kaashoek. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009.

[31] Johannes Stein and Aung Sithu Kyaw. *Irrlicht 1.7 Realtime 3D Engine Beginner's Guide*. Packt Publishing, Birmingham, UK, October 2011.

[32] Paul S. Strauss and Rikk Carey. An object-oriented 3d graphics toolkit. *SIGGRAPH Comput. Graph.*, 26(2):341–349, July 1992.

[33] Kalpathi R. Subramanian and Bruce F. Naylor. Converting discrete images to partitioning trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(3):273–288, July 1997.

[34] Oded Sudarsky and Craig Gotsman. Dynamic scene occlusion culling. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):13–29, January 1999.

[35] L Valente, A Conci, and B Feijó. Real time game loop models for single-player computer games. *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, page 89–99, 2005.

[36] Stefan Zerbst. *3D Game Engine Programming (Game Development Series)*. Premier Press, 2004.