|  |
| --- |
| Wynand Marais |
| Anubis Cookbook |
| Complete Developers Guide |
|  |
| **Wynand Marais** |
| **3/12/2018** |

|  |
| --- |
| This book provides extensive insight into the design of the Anubis game engine as well as significant information for developers seeking to use it. |

Contents

[Chapter 1 – Building from Source](#_Toc509089346)

[Dependencies 1-1](#_Toc509089347)

[Build Options 1-1](#_Toc509089348)

[Other Options 1-2](#_Toc509089349)

[Chapter 2 – Mathematics](#_Toc509089350)

[Chapter 3 – Mathematics](#_Toc509089351)

[Coordinate System 3-1](#_Toc509089352)

[Vectors 3-1](#_Toc509089353)

[Matrices 3-2](#_Toc509089354)

[Chapter 4 – Graphics Module](#_Toc509089355)

[Important Classes 4-1](#_Toc509089356)

[Level of Detail (LOD) 4-2](#_Toc509089357)

[Visibility Calculation 4-2](#_Toc509089358)

[Frustum Culling 4-3](#_Toc509089359)

[Occlusion Culling 4-3](#_Toc509089360)

[Important Classes 4-4](#_Toc509089361)

[Best Practices 4-4](#_Toc509089362)

[Mesh Generators 4-6](#_Toc509089363)

# – Introduction

## Architecture

The Anubis engine has been design to be multithreaded and take advantage of the significant parallel processing capability of modern CPUs. Consequently the engine requires at least three threads to operate as designed. These threads are:

Physics Thread – This is where all the physics computations are performed.

Ready

Ready

Ready

Ready

Ready

Ready

Physics Thread

AI Thread

Main() Thread

In order to remove unwanted dependencies on the Graphics and Sound Module for dedicates servers; the scene graph is not a function of the Graphics Module but rather those of the Physics Module

The PhysicsContext object starts the physics thread upon creation. This object is double buffered and contains both a back and front Scene object. This allows all the physics calculations to be performed on the back scene object while AI and Render Queue calculations are performed on the front scene object. At the end of each simulation interval, an attempt is made to synchronise the back and front scene objects. If the front scene is not available for synchronisation, the context simply progress with the next simulation interval.

# – Building from Source

This chapter demonstrates how to build the Anubis game engine libraries from source. Focus is placed on building for Windows, Linux and OSX using the GCC (g++, mingw), Clang and Visual Studio compilers.

Building Anubis from source is quite easy on Linux, particularly on distributions with precompiled package libraries like Debian, Ubuntu and Centos; and marginally more difficult on Windows where dependency libraries must be manually installed.

## Dependencies

In order to build from source both tool and library dependencies must be met.

### Software Tools

To build from source, the following software tools are required:

* **Git** – Used for cloning the source code from github.
* **CMake** – Used to generate the required project files for the native build system or IDE that will be used for compiling the libraries (i.e. Visual Studio , XCode and Unix Make).
* A suitable compiler like mingw, g++, clang or cl.

Optional software tools include:

* **Doxygen** – Used to generate documentation for the code libraries.

Other software tools that are very useful include:

* **Virtual Box** – Used to create virtual machines for building software on different operating systems.
* **QEMU** – Used for emulating CPU, particularly can be used to emulate Big Endian hosts on Little Endian hardware.

## Build Options

A variety of build options can be specified to enable and disable the compilation of specific engine modules as well as host features.

### Modules

The Anubis engine is broken into unique modules that encapsulate unique functionality, for example, Mathematics, Physics, Graphics, Documentation, etc. By default all the modules are built and must be explicitly disabled if not desired. The list of modules, their dependencies and their respective CMake options is listed in Table 1 below.

Table 1 - CMake options to enable / disable Anubis modules.

|  |  |  |
| --- | --- | --- |
| **Module** | **Dependencies** | **Description** |
| Common |  | Provide common functionality required by all other modules. This module is always built since all other modules are dependent upon it. |
| Math | Common | Provide the mathematical functionality required by the graphics and AI modules. This module is enabled by default and can be disabled using the ANUBIS\_BUILD\_MATHS  option. I.e.  **cmake –DANUBIS\_BUILD\_MATHS=No** |
| Physics | Common  Math | This module is enabled by default and can be disabled using the ANUBIS\_BUILD\_PHYSICS option, i.e.:  **cmake –DANUBIS\_BUILD\_PHYSICS=No** |
| Graphics | Common  Math | Provide the rendering functionality required to draw scene using OpenGL. This module is enabled by default and can be disabled using the ANUBIS\_BUILD\_GRAPHICS option, i.e.:  **cmake –DANUBIS\_BUILD\_GRAPHICS=No** |
| AI | Common  Math  Physics | Provide functionality for common AI functions such as path finding and decision making. |
| Examples |  |  |
| Unit Tests |  |  |
| Benchmarks |  |  |

## Other Options

Other options relating to CPU extensions and specific code libraries can also be applied. The table below lists all the supported options. It is not recommended to change these options if the engine is compile the current host since the best options will be selected by default.

|  |  |
| --- | --- |
| **Option** | **Description** |
| SSE 4.1 | The mathematics libraries can be compiled with SSE4.1 support. The CMake script will check:   1. Whether the required SSE headers are available and the compiler can compile the application. 2. Whether the compile application can be successfully executed.   If both conditions are met, then SSE is enabled by default. If only the first condition is met, then CMake will disable SSE support, however the user can enable it (of course knowing that the application will likely crash when trying to execute on a CPU without SSE support). If neither condition is met, then CMake will disable SSE support and not allow the user to overwrite it (since the required headers are missing or the compiler can’t generate the code, it does not make any sense to allow developers to overwrite this choice).  In the cases that the developer can explicitly set the SSE state, it can be done using the ANUBIS\_ENABLE\_SSE variable, i.e.:  **cmake –DANUBIS\_ENABLE\_SSE=OFF** |

# – Anubis Common

## Identifiable and Sub-Objects

Since the Anubis is thread and can in large manner of configuration which can either use or not use all or one of the modules, a method of establishing relationships between object across all the module domains (Physics, Graphics, Sound, Network, etc) had to be established. This led to the introduction of two concepts:

1. **Identifiable Object (IdentObj)** - Some object must be identifiable across all the module domains and this identity shall be established using UUIDs.
2. **Sub-Objects (SubObj)** - Some objects will form part of a larger identifiable object and must be able to identify the object they belong too. For example, a Game Object can consist of MeshRef, Sound(s), Physics Objects, etc.

In order to realise this paradigm, the following identifiable and sub-objects have been defined:

Sub-objects in Graphics Module:

* Camera

Sub-objects in Physics Module:

* Scene::Node – Since
* IdentObj – A class that provide the

# – Mathematics

This chapter addresses the mathematical implementation of the Anubis Engine. Mathematics is purposely introduced early in this book given the importance of knowing the assumptions made and their implications on calculations. With exception of Quaternions, only high school level mathematics is required to understand the contents of this section.

## Coordinate System

Anubis uses a right handed coordinate system. It is assumed that the Y axis is up, the X axis is left and the Z axis is coming out of the screen.

|  |  |
| --- | --- |
| +Z  +X  +Y | This implies the following identities:  Anubis provides the following unit vectors definitions representing the coordinate system axes:   * Anubis::Math::kXAxis * Anubis::Math::kYAxis * Anubis::Math::kZAxis |

## Vectors

Anubis uses column vectors consisting either two floating point values (2D Vectors) or four floating point values (4D Vectors). This format allows for efficient SIMD optimization by maintaining 128bit data alignment and allowing for a maximum number of vectors to be packed into 128, 256 and 512 bit SIMD registers. This implies that Vector (V) multiplications with matrices (M) are always performed in the order:

|  |  |  |
| --- | --- | --- |
| **2D Vector:** | **4D Vector:** | **Multiplication:** |

It should be noted that when using 4D vectors, the W coordinate must be set appropriately depending on whether the vector indicates a direction or a position. The rules are:

* Set W = 0 for a direction.
* Set W = 1 for a position.

The mathematical proof can be observed simply by trying both. Please refer to the next section, labelled Matrices, for the layout of a translating matrix (the matrix that is used here).

|  |  |
| --- | --- |
| **Translating a Vector with W = 1**  Let: and  Thus:  Results:   * If the vector indicates a direction, it can be seen that W = 1 is not the correct value since the translation operation had the effect of elongating the direction vector. This is problematic when using normalized vectors. * If the vector indicates a position, then it can be seen that W = 1 is not the correct value since the position since it had the effect of translating / moving the point. | **Translating a Vector with W = 0**  Let: and  Thus:  Results:   * If the vector indicates a direction, it can be seen that W = 0 is correct value since the translation operation had no effect on the direction of the vector. * If the vector indicates a position, then it can be seen that W = 0 is not the correct value since the position was not translated / moved by the translation matrix. |

It should be noted that that W component of the vectors remain unmodified during mathematical operations. In the event that an operation returns a vector, it will either:

1. Assume the value of the W component of the lhs vector.
2. Assume the correct W depending on whether a point or a direction is expected as result.

## Matrices

Anubis uses 2D and 4D matrices.

4D Matrices consist of 16 floating point values sequentially and tightly packed in memory.

# – Graphics Module

This section describes how rendering is implemented in the Anubis Engine.

## Important Classes

The following classes are important because no rendering can be performed without them. In order to allow headless dedicated servers, scene information is contained in the Physics Module and rendering information is contained in the Graphics Module. This means that for a scene to be renderable Physics and Graphics Objects are both necessary. The following graphics classes are essential:

* **AttributeArray** – Contains the data and name for a single attribute array, for example vertices, normals and colours. It should be noted that indices are not attribute arrays.
* **MixedAttributeArray** – It takes one or many AttributeArray(s) and interleaves the data. To prevent interleaving, create a single MixedAttributeArray per AttributeArray.
* **Mesh** – Contains one or many MixedAttributeArray(s) and is used for rendering.
* **LODSet** – Contains one or many Mesh(es) and is used to determine the appropriate level of detail to render with. All the meshes are expected to be varying level-of-detail version of each other; however this does not have to be the case. During rendering only one mesh in the set will be used for rendering to a particular camera.
* **MeshRef** – Contains a single LODSet and a single world transform for to mesh to be rendered. This class exist purely to allow reuse of the existing LODSets and Meshes.
* **Shader** – The smallest building block of a shader program, it can be a Vertex Shader, Geometry Shader, Fragment shader, etc.
* **ShaderProgram** – Contains one or many Shader(s) that is linked into a single program. At the least a Vertex and Fragment shader is required to render the scene.
* **Material** – Every mesh and LODSet can be rendered with exactly one material only. The material describes the colour information of the mesh and the ShaderProgram used for rendering it.
* **Camera** – Contains a rendering queue that consists of only the meshes that are visible to the camera. Camera’s traverse the scene every frame and rebuild the render queue.

The following physics classes are essential:

* **Scene** – Describes the game scene. It contains only physics objects, not graphical objects. To link graphical objects to physics objects use the Game Object UUID of the nodes in the scene.
* **Bounding Volume** – This is used for visibility calculations. If an object does not have a bounding volume it will be treated as invisible and not rendered.

The rendering process is:

1. Update the world transforms of all nodes in the tree.
2. Calculate the LOD for all the cameras.
3. Calculate the visibility of all the graphical objects.
4. Sort all the transparent faces for all cameras.
5. Render all the opaque meshes.
6. Render all the transparent meshes.

Special Notes:

1. According to the Khronos group: “As a general rule, you should use interleaved attributes where possible”.

## Level of Detail (LOD)

LOD is implemented as part of the LODSet class in which a list of meshes is stored with an associated range. The range specifies the maximum distance between the camera and the center of the mesh. The list is sorted in ascending order of range during object creation. The table below demonstrates this list structure.

|  |  |
| --- | --- |
| **Mesh** | **Range** |
| Mesh A (First In List) | 20 |
| Mesh B | 50 |
| Mesh C | 100 |
| Mesh D | 200 |
| Mesh E (Last In List) | 100 |

To select the appropriate mesh for rendering the distance between the center of the mesh and the camera is calculated. Then iterating through the list in ascending order, the calculated distance is compared to the range. The mesh is selected by evaluating the following conditions:

1. For the current list item:
   1. If the calculated distance is less than the range, then the current item is used for rendering.
   2. Else if distance is greater than the range, then test the next item.
2. If then end of the list has been reached and no suitable item has been selected, then use the last element in the list.

It can be observer that a mesh will always be selected for rendering. It is worth noting that the LOD selection performed here does not determine if the mesh visibility, it simply selects the appropriate level of detail.

## Visibility Calculation

The visibility calculations consist of two methods:

1. **Frustum Culling** – Calculate the objects that are visible in the camera’s frustum. Frustum Culling is enabled and disable per camera.
2. **Occlusion Culling** – Calculate the objects that are not hidden by other objects in the scene. Occlusion Culling is enabled and disabled per object.

Note that Frustum Culling is always performed first followed by Occlusion Culling, because Occlusion Culling is only concerned with the objects that are visible in the camera’s frustum. Furthermore, Frustum Culling and Occlusion Culling do not make sense to use in all applications. I.e. both have a cost overhead in calculation and both techniques work best on large or computationally complex scenes (from a rendering perspective).

## Frustum Culling

There are many ways to do frustum culling and this is but one way and it is quite simple to understand in implement.

1. Transform the bounding volume into world space.
2. Check if the object is between the camera’s near and far plane by either:
   1. Checking if any of the bound volume’s vertices are between the near and far plane. This method produces the most realistic render compared to method (b) because the whole bounding volume is considered, however it can result in more geometry being rendered. This method is best for games where atmosphere is important and glitch graphics breaks the atmosphere.
   2. Checking if the center of the bounding volume is within the camera’s near and far plane. This method will stop some geometry from rendering though it is technically visible, however it much faster and best scenes with a lot geometry (i.e. RTSes with many units).
3. Project the bounding volume onto the camera’s plane and calculate the bounding area, comparing it with the camera’s bounding area to see if it overlaps. This can be done in one of two ways:
   1. Calculate if the projected 2D triangle of each bound volume and check if it overlaps the camera’s bounding area. This is quite expensive but provides the most accurate rendering result. Unless there is potentially huge benefits to be gained from rendering less geometry, the computational complexity is not worth incurring, use method (b).
   2. Calculate the 2D bounding box of the projected bounding volume. This result in a simple 2D rectangle intersects calculation.

## Occlusion Culling

1. Sort the visible opaque objects by depth using the center of the bounding boxes.
2. For each object in the scene:
   1. Render the object’s occlusion mesh to a lower resolution depth buffer.
   2. Subtract the previous depth buffer.
   3. If it’s different:
      1. Set the new Depth Buffer as the Previous Depth Buffer.
      2. Set the current object as visible.

Basic Example

## Important Classes

The following classes are core to the operation of the rendering system.

### Anubis::Rendering::MeshRef

This class is very important to the rendering process and fulfills three main roles:

1. It provides a method of referencing a single mesh multiple times. Thus a mesh only have to be created on the video card ones, and can be shared an unlimited number of times.
2. It provides a method for specifying mesh Level-of-Detail (LOD).
3. It provides a method for tagging a mesh as visible to a particular camera.

## Best Practices

* Use as few materials as possible.
* Avoid transparent materials or don’t use more than one.

The easiest way to write a game math’s library is to verify it in either Matlab or Octave. Concerning linear algebra, particularly vectors and matrices, several decisions have to make early on:

1. Row or Column Vectors?
   1. Row Vectors assume a horizontal layout with the corresponding transformation matrix on the right hand side.
   2. Column Vectors assume a vertical layout with the corresponding transformation matrix on the left hand side.

Frustum Culling

There are many ways to do frustum culling and this is but one way and it is quite simpler to understand in implement.

1. Transform the bounding volume into world space.
2. Check if the object is between the camera’s near and far plane by either:
   1. Checking if any of the bound volume’s vertices are between the near and far plane. This method produces the most realistic render compared to method (b) because the whole bounding volume is considered, however it can result in more geometry being rendered. This method is best for games where atmosphere is important and glitch graphics breaks the atmosphere.
   2. Checking if the center of the bounding volume is within the camera’s near and far plane. This method will stop some geometry from rendering though it is technically visible, however it much faster and best scenes with a lot geometry (i.e. RTSes with many units).
3. Project the bounding volume onto the camera’s plane and calculate the bounding area, comparing it with the camera’s bounding area to see if it overlaps. This can be done in one of two ways:
   1. Calculate if the projected 2D triangle of each bound volume and check if it overlaps the camera’s bounding area. This is quite expensive but provides the most accurate rendering result. Unless there is potentially huge benefits to be gained from rendering less geometry, the computational complexity is not worth incurring, use method (b).
   2. Calculate the 2D bounding box of the projected bounding volume. This result in a simple 2D rectangle overlap calculation.

## Mesh Generators

To aid testing, debugging and prototyping a few different mesh generators are provided. The mesh generator can be used by including the header <Anubis/Graphics/MeshGenerator.hpp>.

### Rectangle Generator

The rectangle generator generates the mesh data for an arbitrary aligned rectangle. This function is also used to generate the sides of the box in the box generator. Since any mutually orthogonal axes can be used, **m** and **n** is used to designate the axes. The developer has to explicitly define the **m** and **n** axis vectors as, the size of the rectangle along the **m** and **n** axes and the number of division along the **m** and **n** axis. Figure 1 demonstrates this concept.

|  |  |  |  |
| --- | --- | --- | --- |
| **n**  **m**  Figure 1 - Rectangle model. | Calculating the vertices of the rectangle is achieved by stepping through the **n** and **m** axis respectively and calculating the vertices along the **m** axis, then stepping to the next **n** axis interval and repeating the calculation. Figure 2 shows the order in which the vertices are calculated.   |  | | --- | | /\* Count out all the n steps. The +2 is to account for the vertices on  \* the border of the n axis. \*/  for(size\_t n = 0; n < (nDivs + 2); n++) {  /\* Calculate the current starting position. \*/  Vector4f curVert = origin + nStep \* static\_cast<float>(n);  /\* Count of all the m steps. The +2 is to account for the vertices on  \* the border of the m axis. \*/  for(size\_t m = 0; m < (mDivs + 2); m++){  /\* Store the current vertex in the vertices array. \*/  vertices.push\_back(curVert);  /\* Calculate the next vertex. \*/  curVert += mStep;  }  } |   Once all the vertices are calculated, the face indices are calculated. This is achieved stepping through the **n** and **m** axis and calculating the indexes of the three vertices that form the face using a counter-clock-wise winding order. Figure 3 shows the winding order as well as the order in which the faces are calculated.   |  | | --- | | /\* Calculate all the indices for all the faces. \*/  for(size\_t f = 0; f < faceCount; f++){  /\* The first face. \*/  indices.push\_back(curIndice);  indices.push\_back(curIndice + 1);  indices.push\_back(curIndice + mDivs + 3);  /\* The second face. \*/  indices.push\_back(curIndice + mDivs + 3);  indices.push\_back(curIndice + mDivs + 2);  indices.push\_back(curIndice);  /\* Proceed to the next indice. \*/  ++curIndice;  } | |
| **n**  **m**  Figure 2 - Vertex calculation order. |
| **n**  **m**  1  2  3  4  5  Figure 3 - Indices calculation order. |

It can be shown that for a triangulated rectangle with **j** x **k** divisions, the vertex count () face count () and indices count () can be calculated as:

Using these calculations, the vertex count, face count and indices count of Figure 1 can be calculated as:

These calculations can be verified by manually counting the number of vertices, faces and indices.

Other per-vertex properties can also be generated however they mostly exist and performance profiling purposes. This information includes:

* **Normals** – A single normal is calculated and used for all vertices. Since all the vertices are collocated on the same plane, all the normal are the same.
* **Colours**– A single colour (white) is calculated and used for all vertices.
* **Texture Coordinates**– 2D texture coordinates are generated and are mapped to the **m** x **n** axis in the normalised range 0.0f – 1.0f.

### Box Generator

The box generator generates a cube around / centred at the origin. The footprint of the generator function