# Advanced Tools – Assignment Suggestions

Term 2.3

*Additional Course Manual study year 2020/2021*

*Bachelor Creative Media and Game Technologies (CMGT)*
*School of Creative Technology*

Photo by Daniel Cheung on Unsplash

| | |
|---|---|
| Publication date | 25 February 2021 |
| Version | 1 |
| Module coordinator | Yvens R Serpa (YRE03) |
| Lecturers | Yvens R Serpa (YRE03) |
| CMGT roles | Engineer |

## 1 Assignment Recap

The text from below is a slightly different version as presented in the course manual.

Advanced Tools aims to cover the basics of various Tools/Technologies/Techniques (**TTT**). To pass the module, the student must submit and present the module assignment, achieve at least sufficient in all Rubric's criteria and attend all its requirements. The assignment consists of developing a **prototype** to evaluate a specific Tool / Technology / Technique (**TTT**) related to CMGT. The TTT must be part of one of the selected groups mentioned below:

- **Collision Detection Simulation**
- **Visibility Culling**
- **Embedding Languages**
- **Rendering Techniques**
- **Artificial Intelligence**
- **Game Engines**
- **XR Techniques**

The student is free to select any TTT in the list above and even combine them as he/she wants. After choosing a TTT, the student must write down an Evaluation Proposal consisting of **one** line of text explaining what the student wants to evaluate with the given TTT. The module teacher must approve both the TTT selected and the Evaluation Proposal by the end of the last week of the module (Week 8).

## 2 Assignment Suggestions per Topic

### 2.1 Collision Detection Simulation

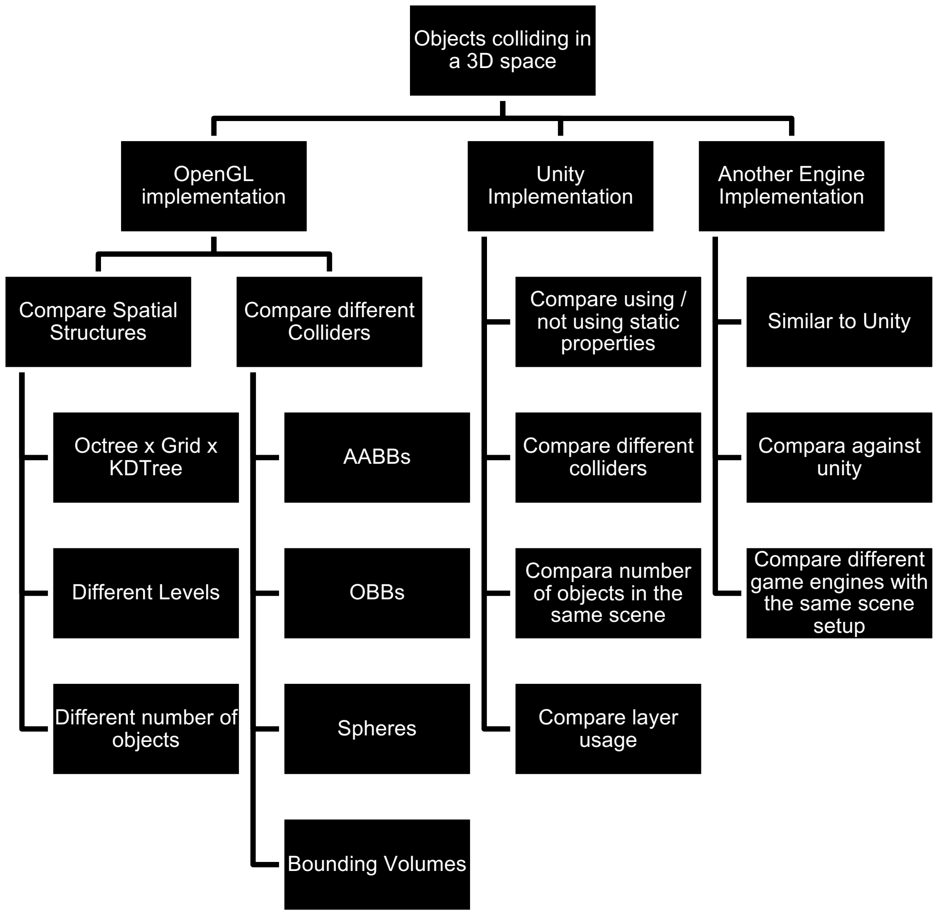

Objects colliding in a 3D space
OpenGL implementation
Unity Implementation
Another Engine Implementation
Compare Spatial Structures
Compare different Colliders
Octree x Grid x KDTree
Different Levels
Different number of objects
AABBs
OBBs
Spheres
Bounding Volumes
Compare using / not using static properties
Compare different colliders
Compara number of objects in the same scene
Compare layer usage
Similar to Unity
Compara against unity
Compare different game engines with the same scene setup

*Figure 1 Collision Detection Simulation ideas.*

You will either program or use tools to evaluate separate collisions in a system and evaluate its performance. Figure 1 shows some ideas for studies to be conducted in the area of Collision Detection.

A standard method to test and validate a collision detection system is to implement a closed environment in which multiple objects collide for a certain number of frames. With that, there are plenty of alternatives to be tested. For instance, an environment with 100 AABBs colliding could be tested using an Octree and a Grid. Then, the FPS could be charted as one aspect of analysis, and the number of tests done per frame could be charted as the second aspect. The properties of the Octree and Grid could also be studied. For instance, you could compare the FPS for an Octree of height three against another one of height 4.

Using a Game Engine, the student can simply use the engine’s physics system to study its performance. Some characteristics, such as the number of collision checks, might be harder to assess due to the engine’s internal working, but then other aspects could be evaluated. For instance, Unity allows objects to be marked as static if they do not move in runtime. Then, the student could assess the performance by using this flag or not on large objects, for instance.

## 2.2 Visibility Culling

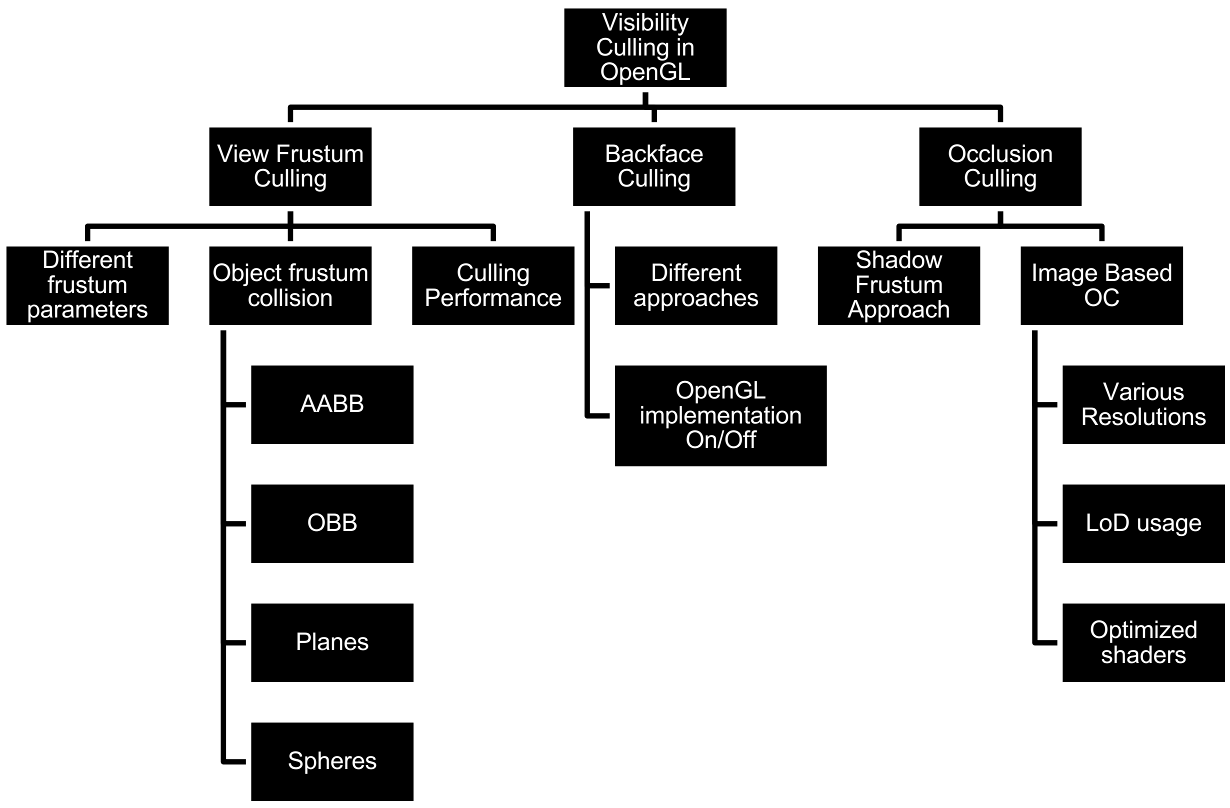


*Figure 2 Visibility Culling ideas*

You will either program or use tools to evaluate different approaches to visibility culling in a system and evaluate its performance. Figure 2 shows some ideas for Visibility Culling studies.

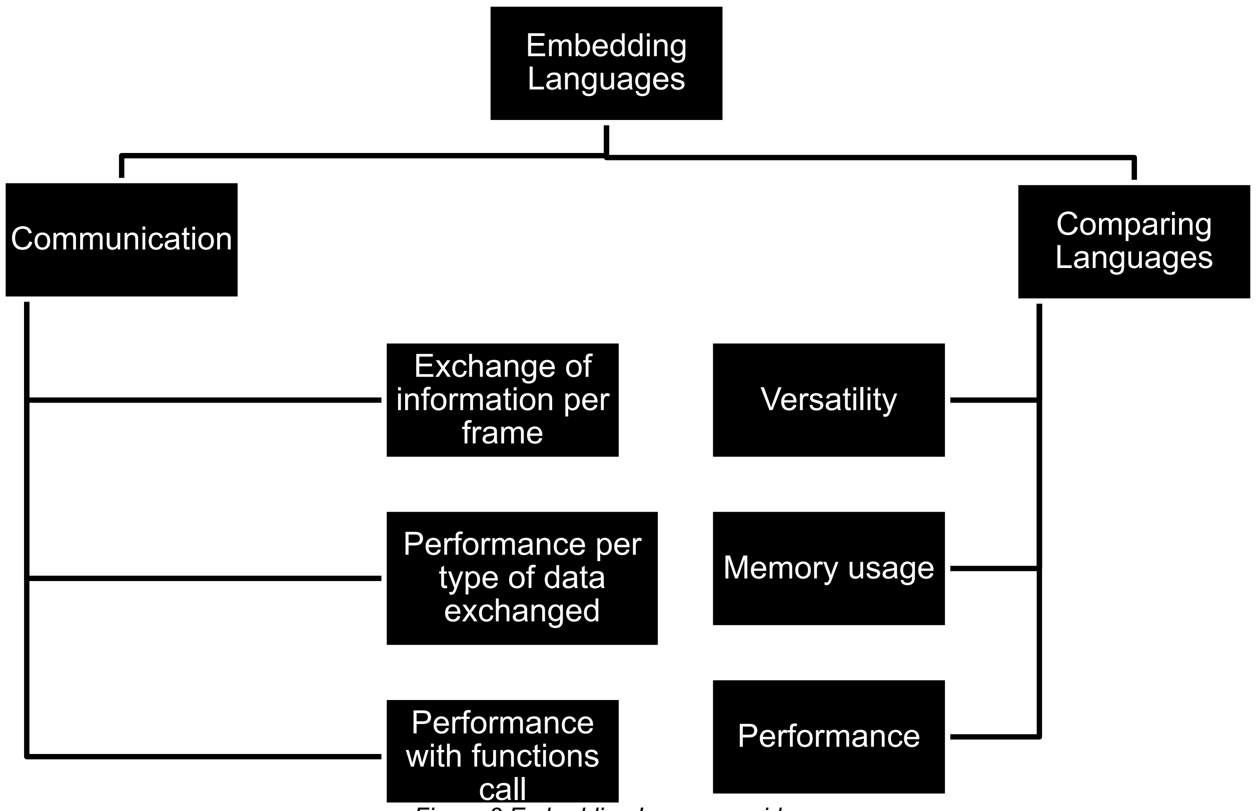
A standard method to test and validate a visibility culling system is implementing a closed environment with multiple objects and a camera moving. Only objects, or, more specifically, their triangles visible by the camera, should be rendered. With that, there are plenty of alternatives to be tested. Apart from the three main algorithms, View-Frustum Culling (VFC), Backface Culling (BC), and Occlusion Culling (OC), the student can test a combination of them and also changes in their respective parameters.

VFC is the easiest method to implement since it is essentially a collision detection between the camera's frustum and the objects in the scene. The student could compare different approaches to check those collisions, as use collision optimization techniques. Instead of testing the objects or their triangles, the student could use colliders and even spatial structures to perform this test hierarchically.

OC is the most challenging and more complex algorithm of the three. Still, there are already more accessible alternatives, such as using OpenGL Occlusion Queries to perform OC in image space (http://www.mbsoftworks.sk/tutorials/opengl3/27-occlusion-query/). In that sense, some of the properties that could be tested are the image resolution for the test, usage of Level of Detail instead of the actual object meshes, and different shaders for OC testing and final render.

## 2.3 Embedding Languages

You will program a system that embeds two different languages (ideally, one of them is an interpreted


Embedding Languages
Communication
Comparing Languages
Exchange of information per frame
Performance per type of data exchanged
Performance with functions call
Versatility
Memory usage
Performance
*Figure 3 Embedding Languages ideas*

language) and evaluates its performance. Figure 3 illustrates some topics that could be studied.

A standard method to test and validate an embedding language is to implement a system with two languages (a base one and another embedded), then analyze properties such as performance, memory usage, versatility (number of possible interactions between languages), and difficulty of embedding (for instance, how many lines of code are required to get a variable from one language to another). It is also a good idea to test with more than one embedding language (Lua and Python, for instance) and compare those aspects.

It is common for some frameworks and APIs to allow scripting with an embedded language. For instance, the code examples for Rendering use C++ and OpenGL with Lua Scripting. Since rendering is an (ideal) real-time application, it is essential to know how complex operations can be performed in a scripting environment to preserve the processing time low and keep the application's processing time in real-time values.

## 2.4 Rendering Techniques

You will either program or use tools to evaluate different types of rendering techniques in a system and evaluate its performance.

There are plenty of different rendering techniques that could be interesting to analyze in their parameters and structures. In general, every method should also be possible to assess the number of objects in the scene (number of triangles) and the number of lights being processed.

Following there's a list of possible effects that could be implemented and analyzed as they seem sufficiently complex:

a) Bloom Effect;
b) Omni Directional Shadows;
c) Tessellation;
d) Image Processing Filters (blur / edge detection / erosion / dilation);
e) Motion blur;
f) Water and Reflections;
g) Deferred Rendering x Forward Rendering;
h) Order Independent Transparency;
i) Gamma Correction;
j) Depth of Field.

Item F is fascinating in the context of multiple lights.

## 2.5 Artificial Intelligence

You will either program or use tools to simulate Artificial Intelligence behavior in a system and evaluate its performance.

There are plenty of different AI algorithms, from Path Finding to Machine Learning, and almost all of them have other parameters to be tested and analyzed, both in terms of performance (processing time, accuracy, memory usage, data, transparency, etc.) and in terms of comparing results among each other. For instance, for Path Finding, you have variations of A* that could be reached and slightly modified to attend specific testing cases. Also, most of those algorithms use heuristics to (try to) optimize their processing time, and those are also subject to experimentation.

Also, given the source code in Lecture 4, the student can play with utility functions and genetic algorithms (more precisely, a stochastic beam search) to assess the performance of different approaches on the simulation presented and student-made simulations and games.

As for Machine Learning, there are various alternatives online, but it is recommended to explore the usage of UnityML (https://unity3d.com/machine-learning). The student can assess different agents' performance, variations on their parameters, and variations in the evaluation scenario. The charts could be built to increase the number of elements (obstacles, other agents, and other factors that interact directly/indirectly with the agents).

## 2.6 Game Engines and XR Techniques

As a broader topic, both Game Engines and XR Techniques are subjects for experimentation on various platforms or comparing them in performance, complexity (how hard it is to have a basic prototype), visual quality, and memory usage. The overall idea is to experiment with simple prototypes using various tools and see how they can achieve the same results. This topic is also easily merged with any of the others above. Figure 4 shows some ideas for Game Engine prototypes.

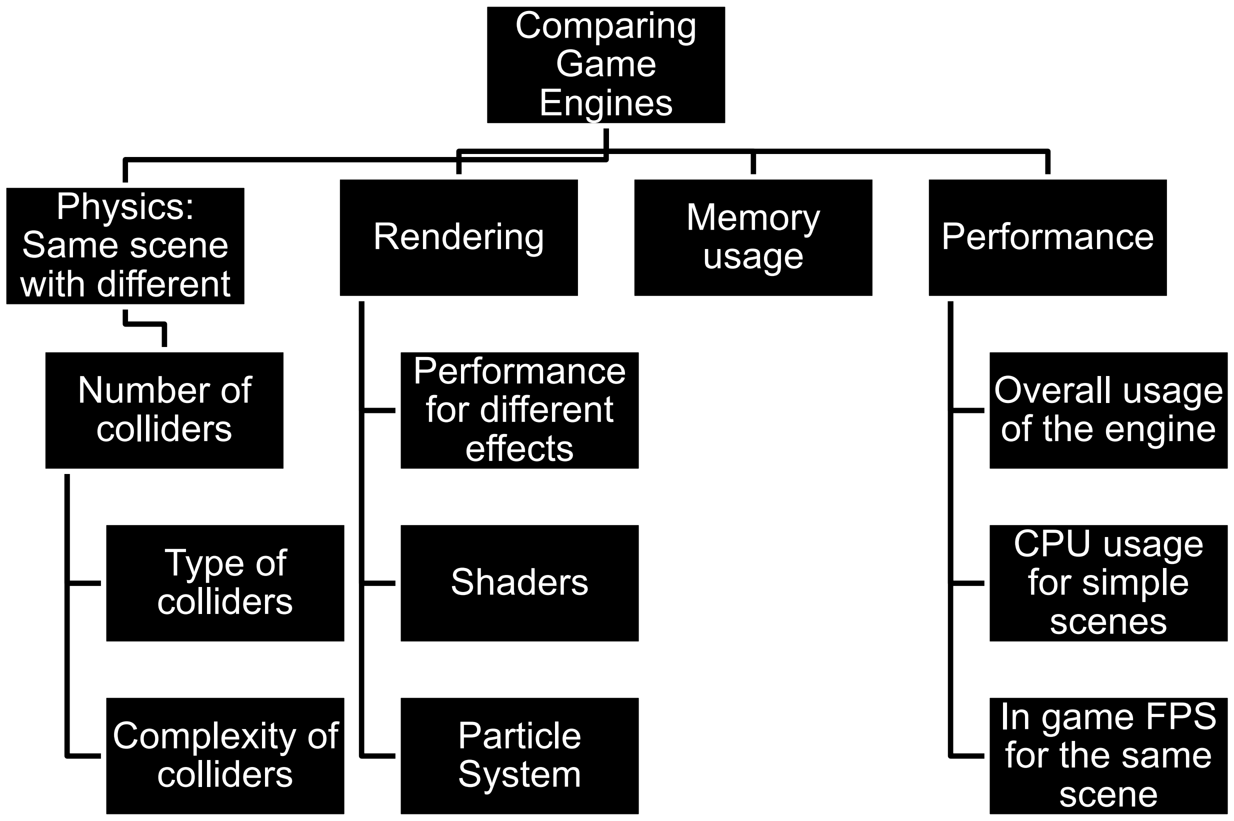

Comparing Game Engines
Physics: Same scene with different
Number of colliders
Type of colliders
Complexity of colliders
Rendering
Performance for different effects
Shaders
Particle System
Memory usage
Performance
Overall usage of the engine
CPU usage for simple scenes
In game FPS for the same scene

*Figure 4 Game Engine ideas*

Another idea for performance comparison in Game Engine is to compare a student-made Game Engine (or part of an engine). As most of the students already worked with some sort of in-house engine for the previous projects, it would be an exciting assessment to assess some of their features and expand on them. For instance, adding Shadows to the 3D renderer or testing the physics engine built for collision detection/collision solving. Apart from that, other non-visual aspects of the engine could also be assessed and analyzed, as the list below exemplifies:

a) Objected Oriented compared to ECS;
b) Manual memory management compared to smart pointers;
c) Compression algorithms for engine-related data;
d) Various external libraries for engine-related features;
e) Audio related algorithms;

As for XR Techniques, the most uncomplicated environment is simply implementing an XR technique using libraries (for instance, a simple 3D environment in VR) and analyzing it. It is a good idea for the assessment to combine it with any of the other groups mentioned before. For instance, how would the physics system behave in terms of FPS in an AR environment? Or how would primary shadow and reflection shaders impact the FPS when done in a VR system?