Geometric algorithms

Project installation

Lidia Ortega Alvarado, Alfonso López Ruiz

Universidad de Jaén

2026

CONTENTS

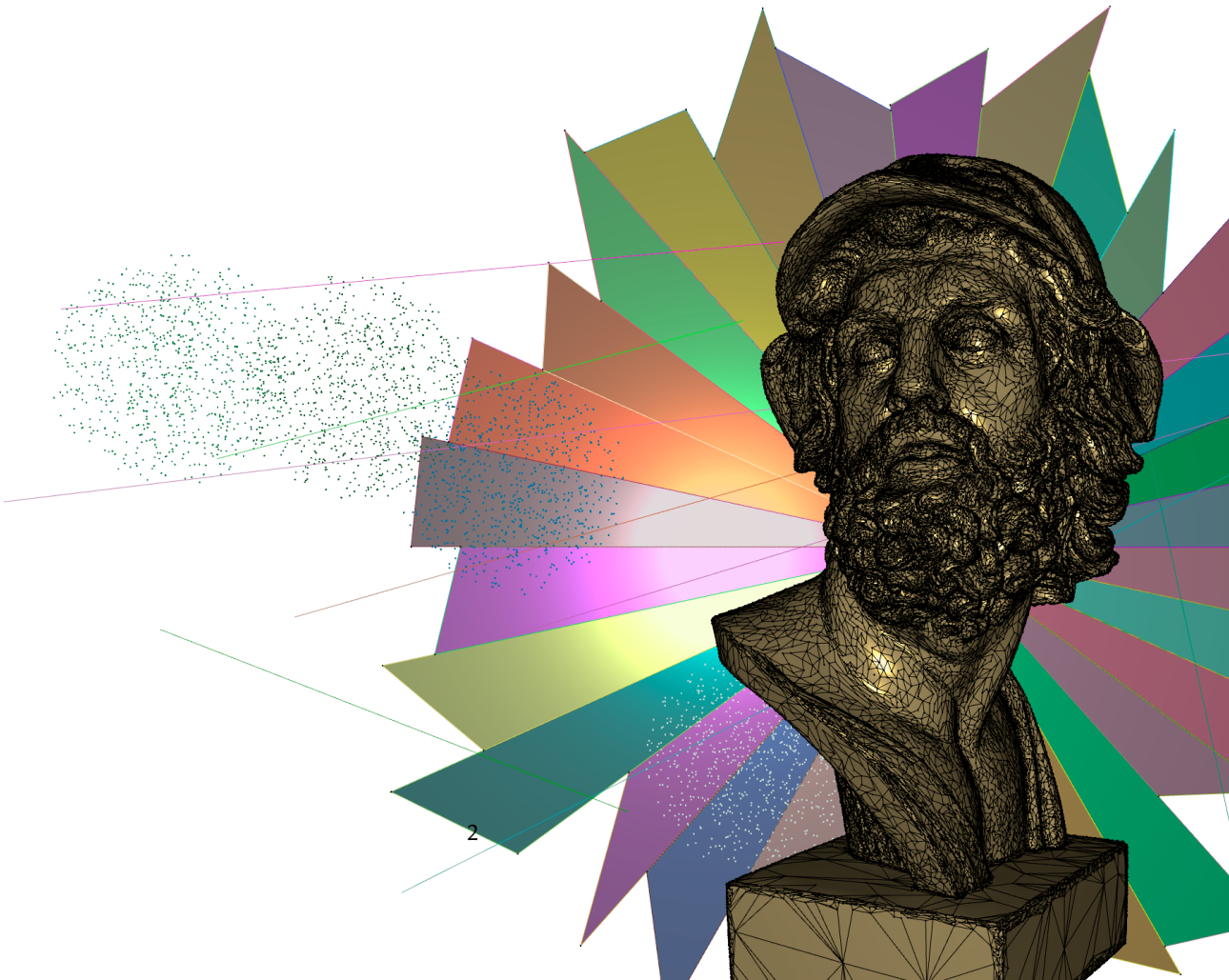
CHAPTER

1 GEOMETRIC ALGORITHMS

The primary objective of this subject is to implement geometric algorithms using 2D and 3D data as well as the spatial data structures that will contain them. Hence, visualizing the outcome of the developed algorithms helps to check whether they adjust to the expected results. However, building a renderer is out of the scope of this subject. This document introduces a baseline renderer that must be configured for an operating system and integrated development environments (IDE) of our choice. The steps to make it work over Windows and Linux operative systems (O.S.) are illustrated using Microsoft Visual Studio and CLion IDEs. We have also tested the Linux steps over Mac O.S. Though it seems to work, it has not been thoughtfully tested.

The baseline code is available at Github and be cloned as follows:

git clone https://github.com/AlfonsoLRz/AlgoritmosGeometricosUJA.git –recurse-submodules



CHAPTER 2

WINDOWS O.S.

Windows O.S. is recommended because of the ease of compilation and management of packages together with vcpkg. We can use several development environments within this O.S. Again, using Microsoft Visual Studio is the most straightforward approach, although you can also find the configuration for the CLion IDE (JetBrains) in section 2.3. This is not intended to be a self-contained manual, and we will assume that Microsoft Visual Studio is already installed. If not, it is sufficient to download a standard version, such as Microsoft Visual Studio Community, and continue the installation with the basic C++ compilation tools.

This section is structured as follows: 1) installation of libraries using vcpkg and 2) configuration of the IDE (either MSVC or CLion).

2.1 Installing vcpkg

vcpkg is a package manager for C/C++ that enables installing, updating and removing libraries, making them globally accessible for environments such as CLion or Microsoft Visual Studio. An installation guide can be found in the vcpkg: Get Started page, although there are two scripts available in the vs folder: `vcpkg_install.bat` and `vcpkg_libraries.bat` which are intended to simplify the download and installation.

> **Nota** **CLion and vcpkg integration**
>
> *CLion has recently integrated vcpkg in case you have not installed it locally on your computer. Be aware that it has not been tested. Guide: vcpkg integration*

First, execute the `vcpkg_install.bat` script to clone the repository of vcpkg, install it, and integrate it with Microsoft Visual Studio. By default, it will place vcpkg in `C:/`, but the script can be updated if you prefer another path or there is no `C:` storage unit. In any case, it is recommended to use an easily accessible path. Once the `.bat` file has been executed, a few environment variables should be included for easily executing vcpkg in the command console, and automatically selecting the 64-bit version during installations. The contents of the mentioned script are as follows:

```
1  cd C:/
2  C:
3  if not exist "vcpkg" git clone https://github.com/Microsoft/vcpkg.git
4  cd vcpkg
5  ./bootstrap-vcpkg.bat
6  vcpkg.exe integrate install
```

Listing 2.1: Installation of `vcpkg`.

Second, the vcpkg folder must be added to the system path, so that the `vcpkg.exe` executable can be globally accessible. To do this, go to `System Properties` > `Environment variables` > `Path` and add a new path: `C:/vcpkg` (modify it according to the previous installation). Secondly, we need to define the architecture of our system so that the installation of packages is carried out with such an architecture.

The architecture can be configured via `System Properties` > `Environment variables` > `New Variable`, using the values in Figure 2.1. Nevertheless, the version and architecture can be passed as an argument during installations.

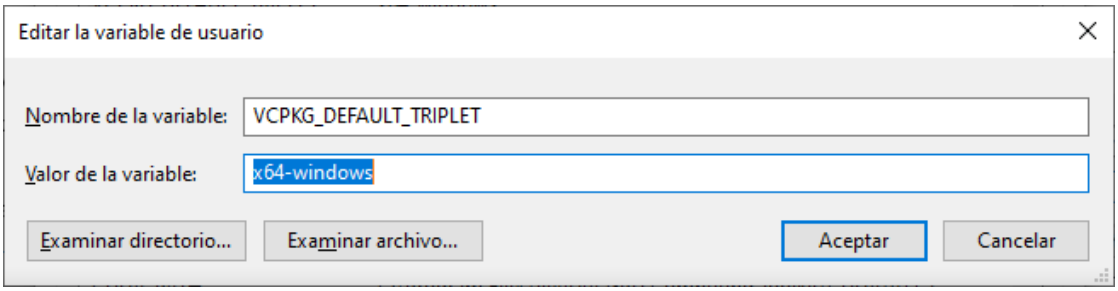


Figure 2.1: Configuration of the preferred architecture for packages to be installed.

We can check that the variables have been registered correctly using the following commands:

Readout of environment variable

```
echo %PATH%
echo %VCPKG_DEFAULT_TRIPLET%
```

In either case, the `vcpkg_libraries.bat` script is prepared to indicate that the desired architecture is 64-bit. A few of the most noteworthy operations of vcpkg are listed below, although we will only use the `install` operation.

- **Search**: vcpkg search name
- **Installation**: vcpkg install name
- **Deletion of packages**: vcpkg remove name
- **List of installed packages**: vcpkg list
- **Upgrade**: vcpkg upgrade name

The contents of the `vcpkg_libraries.bat` script are as follows:

```
1  vcpkg install glfw3:x64-windows
2  vcpkg install glew:x64-windows
3  vcpkg install glm:x64-windows
4  vcpkg install assimp:x64-windows
5  vcpkg install imgui:x64-windows
6  vcpkg install imgui[opengl3-binding]:x64-windows --recurse
7  vcpkg install imgui[glfw-binding]:x64-windows --recurse
8  vcpkg install imguizmo:x64-windows
9  vcpkg install lodepng:x64-windows
```

Listing 2.2: Installation of packages using vcpkg.

> **Nota** **Installation directory**
>
> *From now on, we will refer to our installation directory as* `AGGDIR`.

2.2 Microsoft Visual Studio

After installing vcpkg and cloning the project, open it from the folder `AGGDIR/vs` using Microsoft Visual Studio. The development environment should be restarted if it was open during the installation of vcpkg. Once the project is opened, notice that the toolset of our development environment may not match that of the downloaded project (e.g. v142). In that case, you only need to configure the downloaded project to select your toolset, as shown in Figure 2.2.

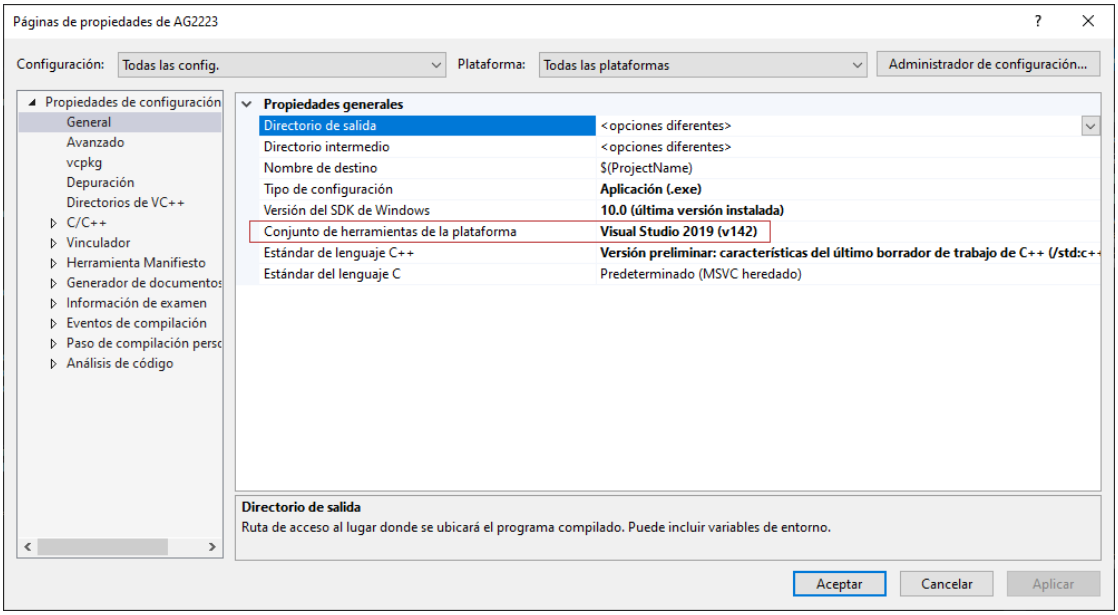


Figure 2.2: Project properties in Microsoft Visual Studio. If the **v142** platform tools are unavailable, we must select other available tools through the drop-down menu.

Once configured, you can compile and run the project.

IMPORTANT: Ensure that the project architecture (e.g., x64) matches the architecture used for vcpkg during the installation process (see Figure 2.3).

Figure 2.3: Configuration of the build architecture in Microsoft Visual Studio.

2.3 CLion

An alternative to Microsoft Visual Studio is the CLion IDE, which can also use the Microsoft Visual Studio compiler. Remember that a one-year educational license can be accessed via the student account (see Figure 3.1). The compilation flow is here guided by the `CMakeLists.txt` file available in the folder `clion`. The following procedure considers that users have downloaded, installed, and successfully launched CLion. The first step is to open the project in `clion` with the IDE. Although the name of `CMakeLists.txt` matches the standard name, it may not be selected by default. On the first boot, you can choose the location of this file or right-click over the file and select `Load CMake project`.

Unlike Microsoft Visual Studio, integrating vcpkg with CLion is not immediate. If vcpkg has been installed in the suggested path, `C:/`, the file `CMakeLists.txt` is ready to use. Otherwise, the location

of vcpkg must be updated.

```
cmake_minimum_required(VERSION 3.24)
project(AG_CLion)

set(CMAKE_CXX_STANDARD 20)

add_definitions(-D_ITERATOR_DEBUG_LEVEL=0)
set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} /MDd")
set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} /MD")

set(FONTS ./Fonts)
set(GEOMETRY ./Geometry)
set(LIBRARIES ../Libraries)
set(PATTERNS ./Patterns)
set(PRECOMPILED_HEADERS ./PrecompiledHeaders)
set(RENDERING ./Rendering)
set(VCPKG C:/vcpkg/installed/x64-windows)

include_directories(${FONTS})
include_directories(${GEOMETRY})
include_directories(${LIBRARIES})
include_directories(${PATTERNS})
include_directories(${PRECOMPILED_HEADERS})
include_directories(${RENDERING})
include_directories(${LIBRARIES}/ImGuiFileDialog)
include_directories(${LIBRARIES}/lodepng)
include_directories(${VCPKG}/include/)
include_directories(${VCPKG}/include/assimp)
include_directories(${VCPKG}/include/GLFW/include)
include_directories(${VCPKG}/include/gl)
include_directories(${VCPKG}/include/glm)

set_source_files_properties(stdafx.cpp
        PROPERTIES
        COMPILE_FLAGS "-include stdafx.h")

link_directories(${VCPKG}/lib)

set(SOURCE_FILES
        main.cpp
        ${GEOMETRY}/AABB.cpp
        ${GEOMETRY}/AABB.h
        ${GEOMETRY}/BasicGeometry.h
        ...
        ${FONTS}/font_awesome_2.cpp)
add_executable(AG_CLion ${SOURCE_FILES})

target_link_libraries(AG_CLion imgui imguizmo glfw3dll glew32 OpenGL32 assimp-vc143-mt)
```

From now on we will stick with the default configuration of CMake (Figure 2.4: MinGW as the toolchain and Let CMake decide the generator). Notice that other options are available, such as using the Visual Studio compiler or the Debug build type instead of Release (the Release executable is faster, at the expense of being unable to trace the program flow and having a bit slower compilation). The next step is compiling the project or recompiling it if it was previously compiled and failed. There is also a Clean

option for removing data generated during build.

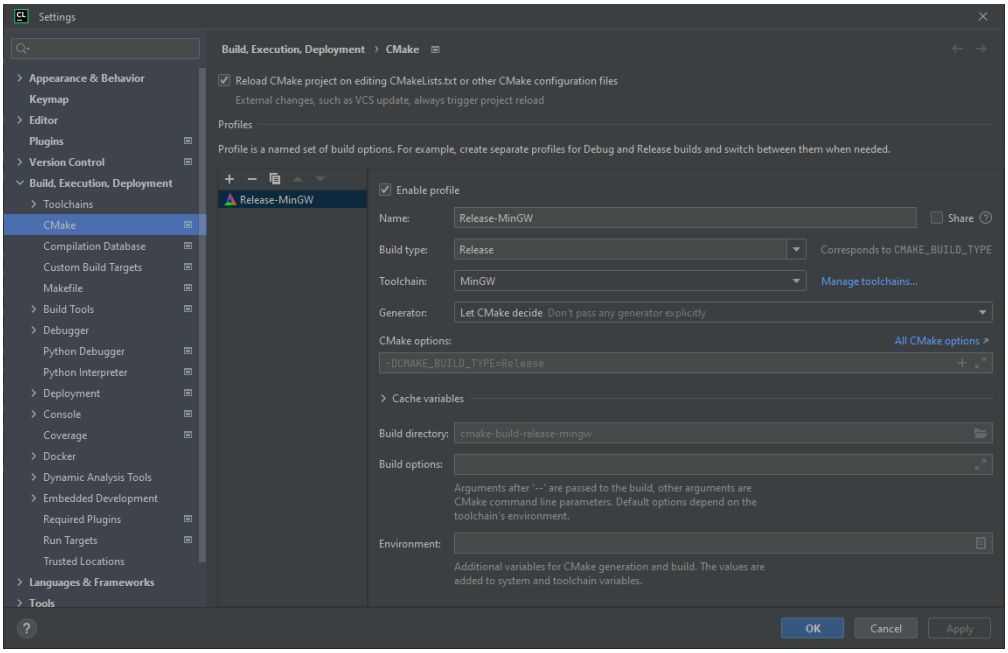


Figure 2.4: CMake configuration on Windows.

The outcome of the compilation is a `.exe` file in `cmake-build-debug/Debug/` or `cmake-build-release/Release/`. As we integrated vcpkg manually into our project, there are missing `.dll` files that will initially make the execution fail. We have already prepared two scripts that 1) copy the required `.dll` files from our vcpkg installation, and 2) copy the `vs/Source/Assets` folder with the shaders and required fonts. These scripts have been named `prepare-debug.bat` and `prepare-release.bat`. They check whether the output directory is already created, and therefore, can be launched even before compiling.

The contents of the `prepare-debug.bat` script are as follows:

```
@echo off

:: Move to the folder where this script is located
cd /d %~dp0

set VCPKG_DIR=C:/vcpkg/
set TRIPLET=x64-windows
set OUTPUT_DIR=cmake-build-debug/Debug/

set DLL_LIST="glew32.dll" "glfw3.dll" "assimp-vc143-mt.dll" "zlib1.dll" "poly2tri.dll" "minizip.dll" "pugixml.dll"

if not exist "%OUTPUT_DIR%" (
    mkdir "%OUTPUT_DIR%"
)

echo Copying selected .dll files from vcpkg to %OUTPUT_DIR%...
for %%D in (%DLL_LIST%) do (
    if exist "%VCPKG_DIR%\installed\%TRIPLET%\bin\%%~D" (
        echo Copying %%~D to %OUTPUT_DIR%
        copy "%VCPKG_DIR%\installed\%TRIPLET%\bin\%%~D" "%OUTPUT_DIR%" >nul
    ) else (
        echo WARNING: %%~D not found in %VCPKG_DIR%\installed\%TRIPLET%\bin
    )
)

echo Done. Selected DLLs have been copied to %OUTPUT_DIR%.
pause

echo Copying Assets folder to %OUTPUT_DIR%...
xcopy /E /I /H /Y "..\vs\Source\Assets" "%OUTPUT_DIR%\Assets"
```

Listing 2.3: Script for copying DLL files and resources into the output folder of the debug build.

Once the program is successfully compiled and the scripts are executed, running the application should

display a result like the one shown in Figure 2.5.

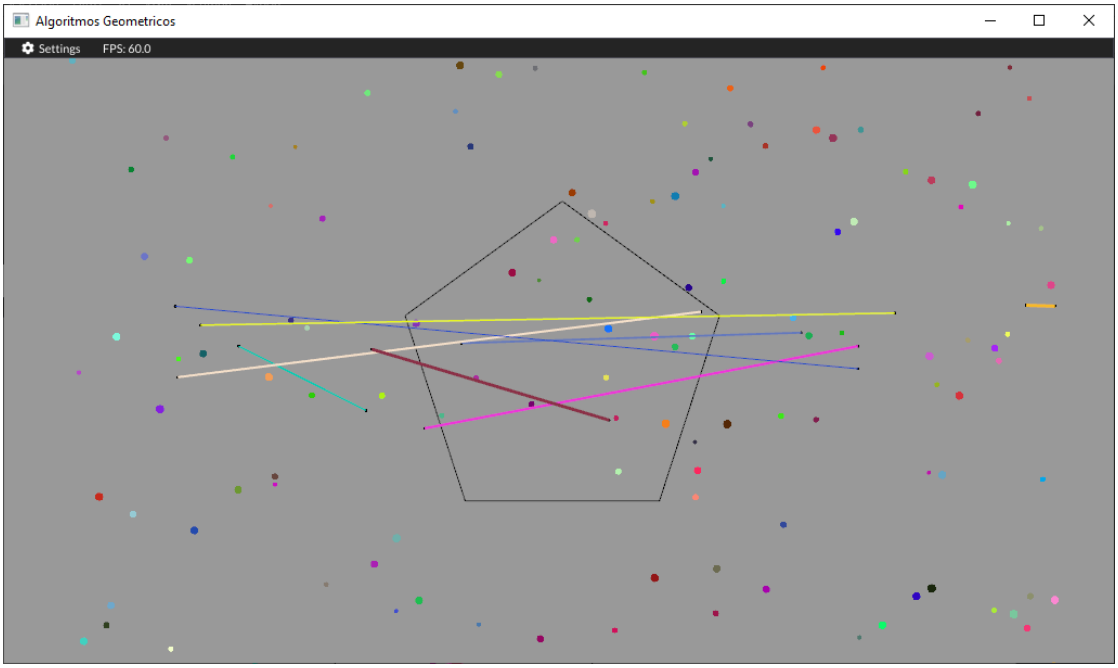


Figure 2.5: Rendering of the default scenario after launching the application with CLion (Windows O.S.).

CHAPTER 3

LINUX O.S.

In the case of Linux, this document specifically focuses on the CLion development environment (JetBrains).

3.1 CLion

Installing CLion on Linux is straightforward by running the command `sudo snap install clion --classic`. Remember that a one-year educational license can be accessed via the student account (see Figure 3.1). Following this, we will install the required libraries. First, clone the repository in the folder of your choice using either HTTPS or SSH, with the first method being the easiest (`git clone https://github.com/AlfonsoLRz/AlgoritmosGeometricosUJA.git`). Once the repository has been downloaded, we can run the script `/install_libraries.sh` in `{AGG_DIR}/clion_linux` after providing execution permissions.

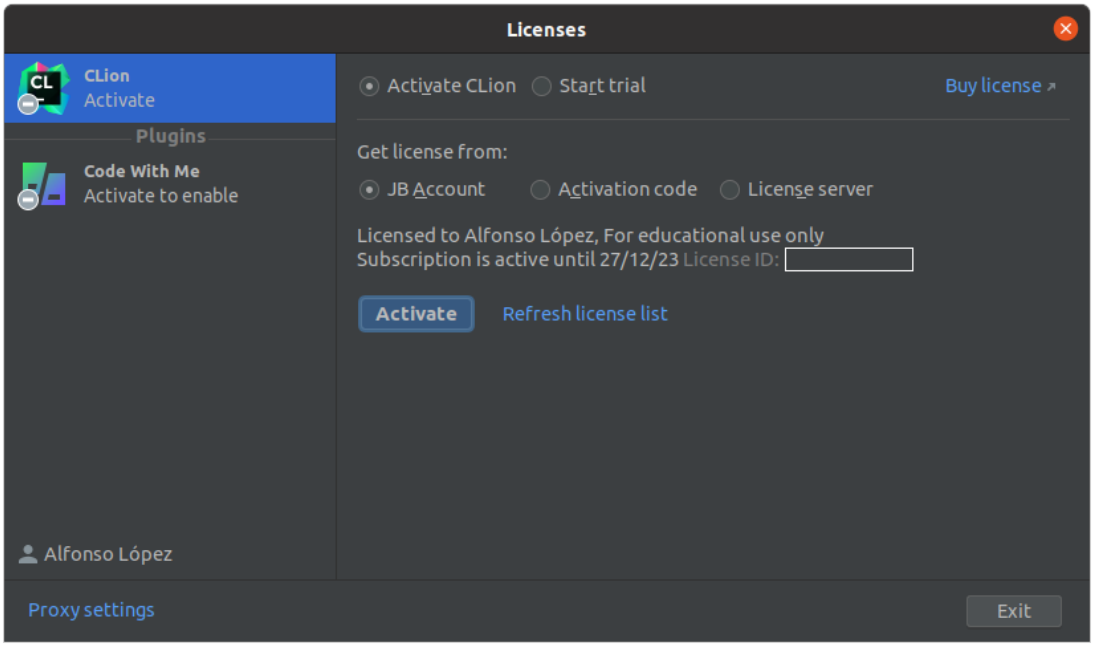


Figure 3.1: CLion license activation via an educational email.

Installation script

```
cd {AGG_DIR}/clion_linux
chmod 777 install_libraries.sh
./install_libraries.sh
```

The contents of this script is shown below, taking into account that we need GLFW (graphical interface), GLM (mathematical operations), GLEW (interface with OpenGL), Assimp (object loading) and TBB

(multi-threaded execution).

```
#!/bin/bash
sudo apt-get install libglfw3-dev
sudo apt-get install libglew-dev
sudo apt-get install libglm-dev
sudo apt-get install libassimp-dev
sudo apt-get install libtbb-dev
```

Listing 3.1: Installation of libraries needed in the project

When the installation is done, we can open CLion and use the menu `File > Open` to open the project located in `{AGG_DIR}/clion_linux/`. The file `CMakeLists.txt` is provided with the project, although we will have to indicate that this will be our roadmap (by default it seems not to be selected, even though the name matches). The rest of the options have been left as they appear by default (see Figure 3.2), except for the type of `Build (Debug, Release)`, which we have selected as `Release` to speed up the execution; however, either option can be used. Configuration of `CMake` must be done via the action `File > Settings > Build, Execution, Deployment > CMake`, where we can configure the build type (`Debug, Release`), which *toolchain* to use and other command line options.

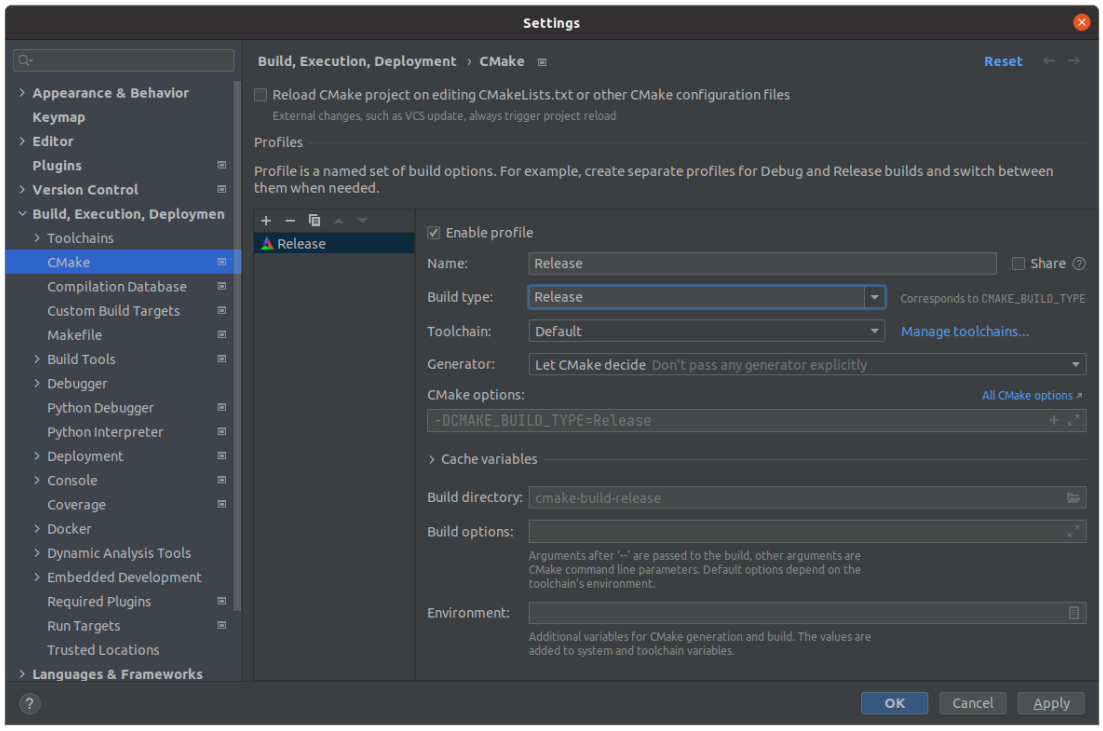


Figure 3.2: CMake configuration window in Linux.

Once the configuration is complete, we can compile the project. If there are no errors, we only need to move the assets folder (`{AGG_DIR}/vs/Source/Assets`) to the directory where the executable is located (`cmake-build-debug` or `cmake-build-release`). To do this, we can copy the resource folder as shown below, considering that we are located in the directory `{AGG_DIR}/clion_linux/`. This task is also automatized in the scripts `prepare-debug.sh` and `prepare-release.sh`.

Copying the resources folder

```
cp ../vs/Source/Assets/ cmake-build-debug/
```

CHAPTER 4

PROJECT FEATURES

The interactions available in the application are described below.

1. **Camera**: interaction through keyboard and mouse.

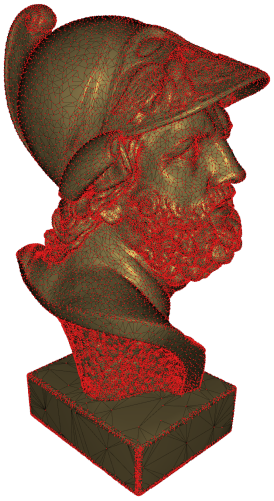
Movement	Interaction
Forward	Right button + W
Backwards	Right button + S
Left	Right Button + A
Backwards	Right-click + D
Zoom	Mouse wheel
Horizontal orbit	X
Vertical orbit	Y
Camera pan	Left mouse button
Reset camera	B

2. **Gizmos**: interaction with models to carry out translation, rotation and scaling transformations. For this, it is necessary to open the menu `Settings > Models` and select a model.

Operation	Interaction
Model translation	T
Model rotation	R
Model scaling	S

3. **Displaying different primitives** that have been generated during the loading of the models in the scene (`SceneContent`).

> **Nota Primitives**
>
> *In this document, the term primitive is used to refer to a basic unit that defines how the vertices of a model are interrelated. For example, a triangle, composed of three vertices, has only one primitive for a mesh of triangles (defined as $\{0, 1, 2\}$). However, if we want to draw our triangle as a wire mesh, composed of lines, we will have three different primitives ($\{0, 1\}, \{1, 2\}, \{2, 0\}$). In this manner, the indices correspond to the position occupied by a vertex in the vector* **`geometry`**.



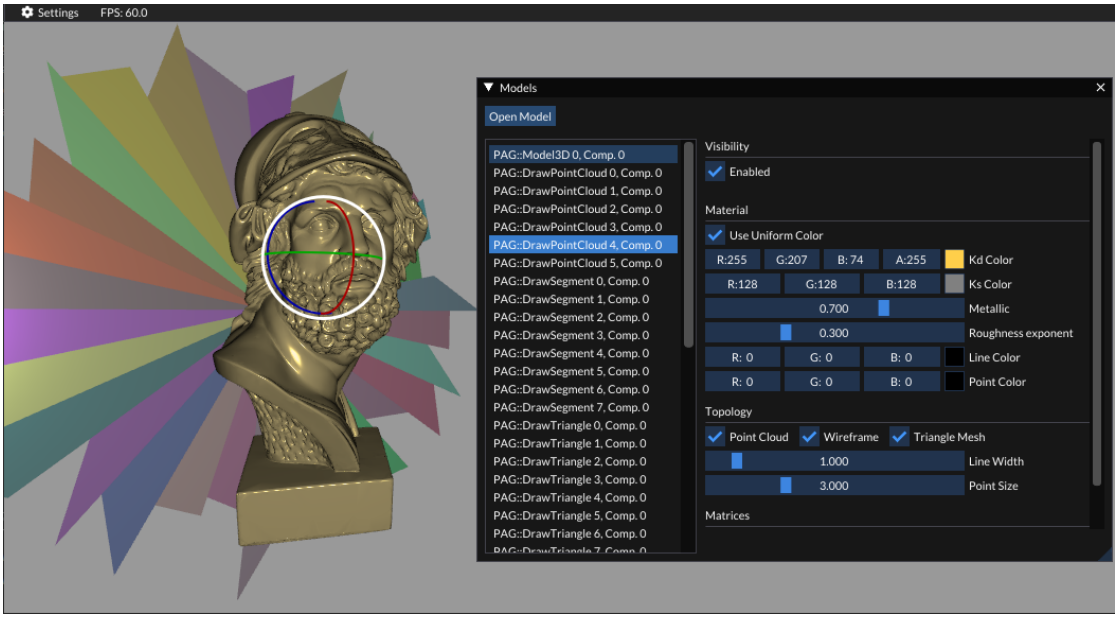


Figure 4.1: Model transformation via the interface. In this case, the image depicts the rotation gizmo.

The primitive to be rendered can be controlled at the global level, so we can enable and disable its rendering in the menu `Settings > Rendering`, or at the local level (for each model) via the `Settings > Models` menu.

Operation	Interaction
Activate/Deactivate point cloud	0
Activate/Deactivate wire mesh	1
Activate/Deactivate triangle grid	2

4. Screenshot with antialiasing (for the documentation `:D`). We can make a screenshot using the keyboard or the GUI (menu `Settings > Screenshot`). The latter option also enables changing the size of the image as well as the system path.

Operation	Interaction
Screenshot	K

Several other functionalities are offered from the interface:

1. **`Settings > Rendering`**:
   - Modification of the visibility of previously revised topologies.
   - Modification of the background colour.
2. **`Settings > Camera`**:
   - Modification of camera properties, including the projection.
3. **`Settings > Lights`**:
   - Modification of the properties of a single point light (colours and position). Note that, the objective of this subject is not to learn the principles of rendering and this point light should be enough to visualize the triangle meshes in the scene.
4. **`Settings > Models`**:

- Modification of transformation matrices of models in the scene.
- Modification of materials (point colour, line colour and triangle mesh colour/default texture).
- Modification of point size and line width.
- Loading 3D models (`.obj`, `.gltf` y `.fbx`).

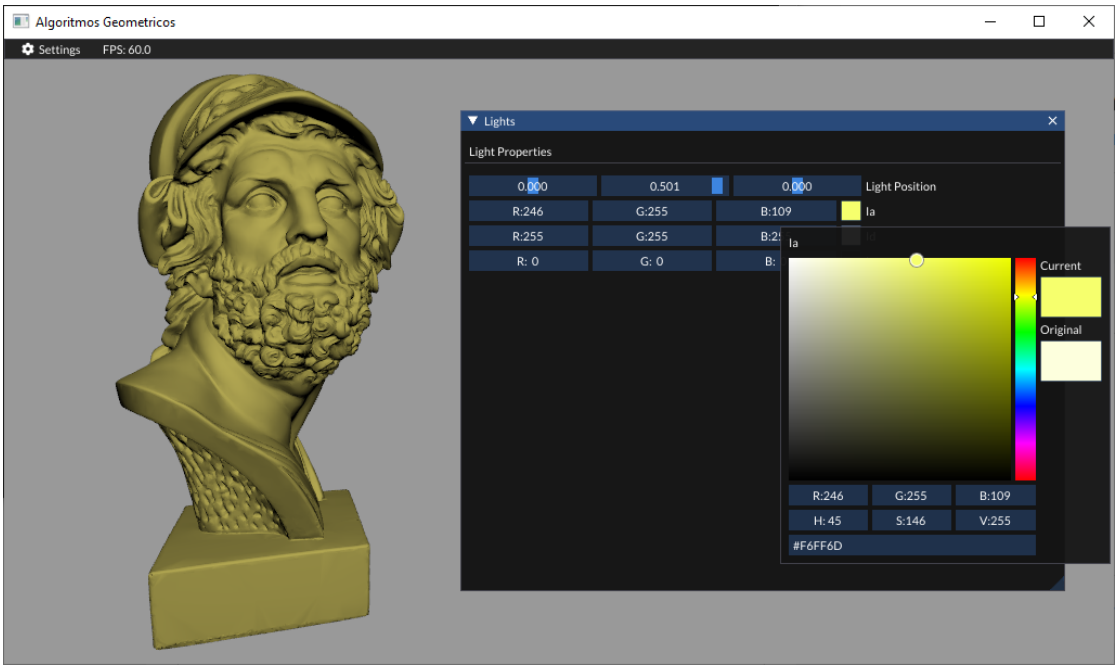


Figure 4.2: Configuration of a point light.

4.1 Integration of new 2D/3D objects

New models can be implemented as a subclass of `Model3D`, which contains all the necessary methods to load and draw objects in the GPU. Therefore, these tasks can be omitted and the remaining work consists of defining the geometry and how it interrelates. Note that the attributes of a vertex (`VAO::Vertex`) are (in strict order): position (vec3), normal vector (vec3) and texture coordinates (vec2). Thus, we can add new vertices to our model using the following syntax:

```
componente->_vertices.insert(component->vertices.end(), { vertices })
```

where vertices can be defined as follows:

```
{
    VAO::Vertex { vec3(x, y, z), vec3(nx, ny, nz) },
    VAO::Vertex { vec3(x, y, z) },
    VAO::Vertex { vec3(x, y, z), vec3(nx, ny, nz), vec2(u, v) }
}
```

Following this strict order is important, although we can omit some of these attributes if they are unknown. Regarding primitives, three vectors (point cloud, wireframe, and triangle mesh) will be available in the variable `component->_indices`. Again, we can insert primitives as shown below:

- **Triangles**:

```
componente->_indices[VAO::IBO_TRIANGLES].insert(
    componente->_indices[VAO::IBO_TRIANGLES].end(),
    {
        0, 1, 2,
        1, 2, 3,
        ...
    })
```

- **Lines**:

```
componente->_indices[VAO::IBO_WIREFRAME].insert(
    componente->_indices[VAO::IBO_WIREFRAME].end(),
    {
        0, 1,
        1, 2,
        ...
    })
```

- **Points**:

```
componente->_indices[VAO::IBO_POINT_CLOUD].insert(
    componente->_indices[VAO::IBO_POINT_CLOUD].end(),
    {
        0, 1, 2, 3, 4
        ...
    })
```

> **Nota** **Fast generation of point clouds**
>
> *Given a number of vertices n, we can generate a vector as* $\{0, 1, 2, ..., n-1\}$ *using* `std::iota(begin, end, 0)` *after* `vector.resize(n)`.

> **Nota** **Automatic generation of some topologies**
>
> *You have available some methods in the class Component to automatically generate topologies from others of a greater entity, taking into account that: "**triangle mesh**" > "wireframe" > "point cloud". Thus, for an OBJ model defined by a set of triangles, the wireframe and point cloud topologies can be generated automatically. Otherwise, point clouds can also be extracted from a wireframe topology.*

In addition, a list of objects is displayed in the menu `Settings > Models`. Due to C++ inheritance limitations, the class name of an object that inherits from `Model3D` cannot be queried in its constructor. However, once constructed, it is possible to obtain its name. For this reason, if we want the objects to have a meaningful name, we can use the function **overrideModelName**.

The methods `SET` of the class `Model3D` have been implemented in such a way that the calls can be chained in the same line after building the object, including operations such as `overrideModelName`, `setPointColor`, `setLineColor` or `setTopologyVisibility`.

To recap, the basic flow of a constructor of a **Draw** class is as follows for most cases:

```
Component* component = new Component;

// Define geometry
component->_vertices.push_back(...);
component->_vertices.insert(component->_vertices, { ... });

// Define primitives
component->_indices[VAO::IBO_TRIANGLE].push_back(...);
component->_indices[VAO::IBO_TRIANGLE].insert(component->_indices[VAO::IBO_TRIANGLE], { ... });

// Automatic generation of other derived primitives
component->completeTopology();

// Calculate boundaries
this->calculateAABB();

// Send component data to GPU
this->buildVao(component);

// Add to the main vector to include in the rendering loop
this->_components.push_back(std::unique_ptr<Component>(component));
```

4.1.1 Scene management

The management of the scene elements will be carried out in RenderCore/SceneContent (Rendering/SceneContent in the file system). For this, we have a method called buildScenario where the scene content will be instantiated only at the start. An example of a scene definition is the following:

```
vec2 minBoundaries = vec2(-1.5, -.5), maxBoundaries = vec2(-minBoundaries);

// Triangle mesh
auto model = (new DrawMesh())->loadModelOBJ("Assets/Models/Ajax.obj");
model->moveGeometryToOrigin(model->getModelMatrix(), 10.0f);
this->addNewModel(model);

// Spheric randomized point cloud
int numPoints = 800, numPointClouds = 6;

for (int pcIdx = 0; pcIdx < numPointClouds; ++pcIdx)
{
   PointCloud* pointCloud = new PointCloud;

   for (int idx = 0; idx < numPoints; ++idx)
   {
      ...
      pointCloud->addPoint(Point(rand.x, rand.y));
   }

   this->addNewModel((new DrawPointCloud(*pointCloud))
                     ->setPointColor(RandomUtilities::getUniformRandomColor())
                     ->overrideModelName());
   delete pointCloud;
}

// Random segments
int numSegments = 8;

for (int segmentIdx = 0; segmentIdx < numSegments; ++segmentIdx)
{
   ...
   SegmentLine* segment = new SegmentLine(a, b);

   this->addNewModel((new DrawSegment(*segment))
                     ->setLineColor(RandomUtilities::getUniformRandomColor())
                     ->overrideModelName());
   delete segment;
}

// Random triangles
int numTriangles = 30;
float alpha = ...;

for (int triangleIdx = 0; triangleIdx < numTriangles; ++triangleIdx)
{
   ...
   Triangle* triangle = new Triangle(a, b, c);

   this->addNewModel((new DrawTriangle(*triangle))
                     ->setLineColor(RandomUtilities::getUniformRandomColor())
                     ->setTriangleColor(color)
                     ->overrideModelName());
   delete triangle;
}
```

To be noted:

- **`addNewModel`** receives a pointer to an object whose class inherits from `Model3D`.
- **`content`** is the scene content.
- The **`setters`** methods of a 3D model have been implemented as `Model3D* set***()` to chain different calls in the same code line.
    - What can be changed through `setters`?:
        - Colour: `setPointColor`, `setLineColor`, `setTriangleColor`. The latter also contemplates the alpha value by receiving a `vec4` value.
        - Primitive size/width: `setPointSize` y `setLineWidth`.
        - Visibility of primitives: `setTopologyVisibility`. It receives the kind of primitive, `VAO::IBO_slots`, and a Boolean value.
        - `moveGeometryToOrigin`: this method calculates the transformation matrix that places a model in the origin of the coordinate system. It even receives a scaling factor to make it fit in the viewport.
        - `overrideModelName`: by default, a model obtains a generic name in the constructor, e.g., `Model3D 8, Comp. 0`. Nonetheless, this name can be automatically customized so that it can be rapidly visualized in the model list (which can be accessed via `Settings > Models`. Note that a subclass cannot query the class name in the constructor, although it can be done once built.

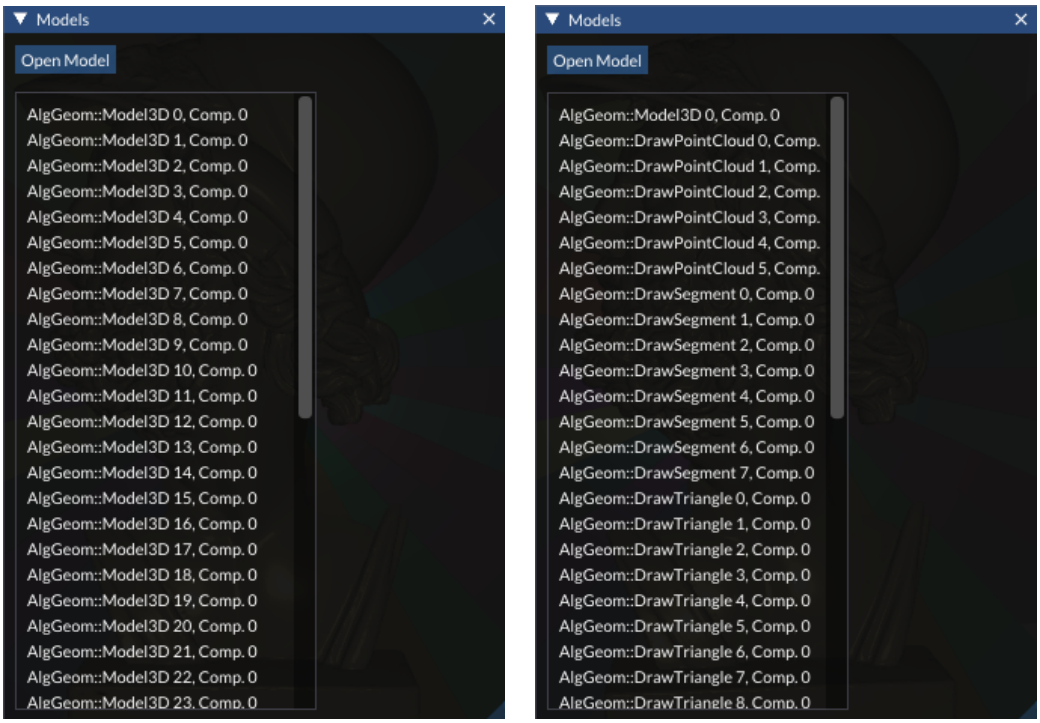


Table 4.1: Comparison of two model lists, using generic names and custom names for each model (automatically assigned).

Finally, note that the camera is configured in the `SceneContent` class, using the `buildCamera` method so that you can modify its position or any other view parameter. During the first assignments, the camera is constrained to 2D by blocking three-dimensional movements. This limitation can be modified in the camera builder. Also, notice that the camera will be built after generating the models, and therefore it is possible to adjust it to the scene content. To achieve this, the camera has a method

track to pass an axis-aligned bounding box of any 3D model (it could be the main model in the scene or the boundaries of the whole scene).

4.1.2 Multiple instances

Most primitives in this subject have a small footprint (segments, rays, circles, etc.). The main source of problems in terms of performance comes from rendering thousands and millions of these instances. This is easily observed in voxelizations, i.e., discretizations of primitives into quads and cubes (2D/3D). For instance, a small voxelization of 256^3 implies the rendering of sixteen million cubes (x6 quads each). These scenarios degrade the performance of our renderer and harden the interaction. For this reason, we have wrapped the rendering of voxelizations into two classes: `Voxelization` and `DrawVoxelization`.

The `Voxelization` objects could be generated explicitly, for instance, to build a sphere, but they will be mainly extracted from existing objects (e.g., triangle meshes). The implementation under `DrawVoxelization` is based on OpenGL's instancing. Instead of drawing isolated objects, all of them are rendered in the same batch. Together with their geometry, we can pass information such as translations, rotations, etc. that relate to each instance individually. In this manner, we can render millions of equidistant cubes with the same size.

Following is the code to generate a `DrawVoxelization` instance from a triangle mesh. Unlike the previous primitives, most of the required code to make voxelizations work is not in the repository. This is not considered baseline code and should be implemented by the students.

```
1  void AlgGeom::SceneContent::buildScenario()
2  {
3      ...
4
5      // Mesh
6      TriangleModel* triangleModel = new TriangleModel("Assets/Models/Ajax.obj");
7      auto model = (new DrawMesh(*triangleModel))->overrideModelName();
8      this->addNewModel(model);
9
10     // Voxelization
11     const vec3& voxelSize = vec3(.015f);
12     Voxelization* voxelization = new Voxelization(triangleModel, voxelSize);
13     voxelization->sweep();
14     voxelization->printData();
15
16     this->addNewModel(voxelization->getRenderingObject(true)->setModelMatrix(
17       glm::translate(mat4(1.0f), vec3(-.2f, .0f, .5f)))->overrideModelName()
18   );
19
20     delete triangleModel;
21     delete voxelization;
22 }
```

We can either build a voxelization from triangle meshes or build our models with implicit functions (e.g., a sphere), as shown in Figure 4.3.

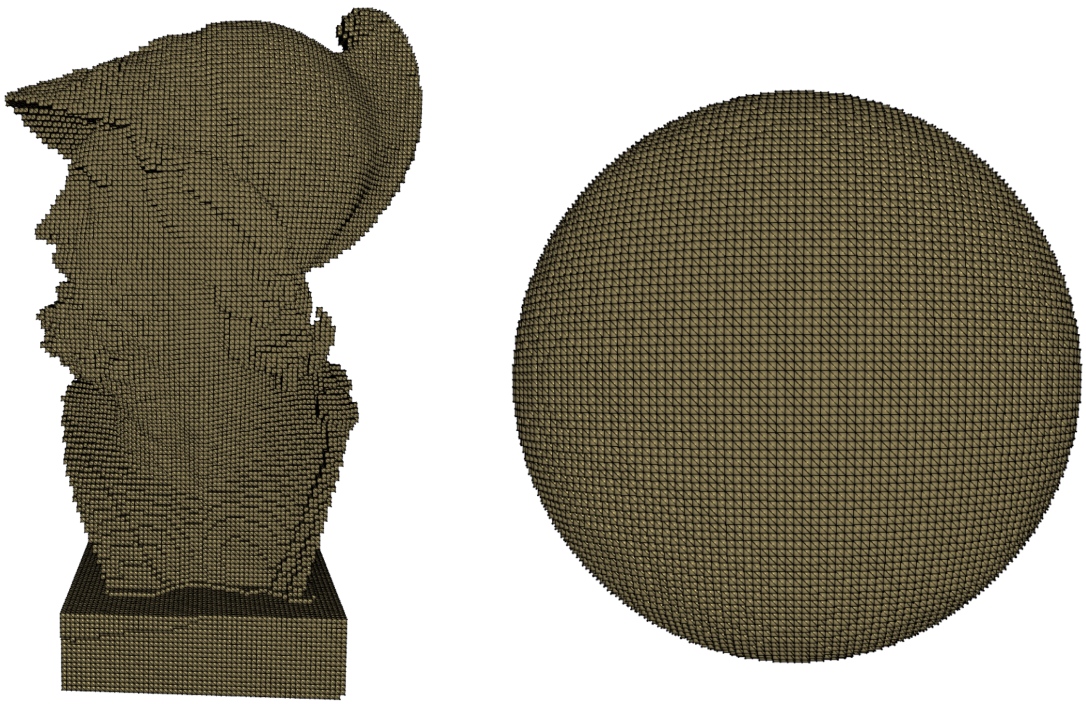


Figure 4.3: Examples of voxelizations in our renderer.