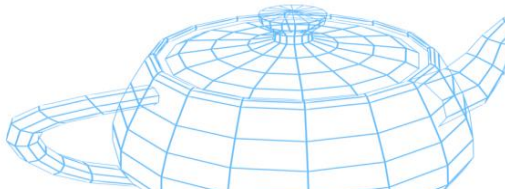
SUPSI

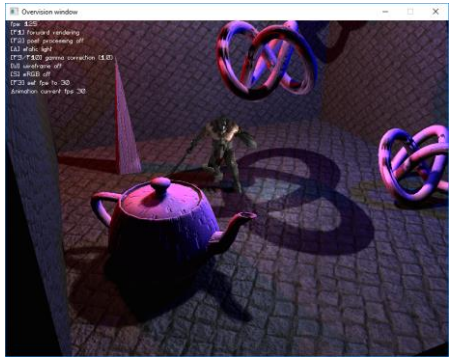
Computer Graphics

3D Graphics Engines (1): basic architecture

Achille Peternier, adjunct professor



The big picture



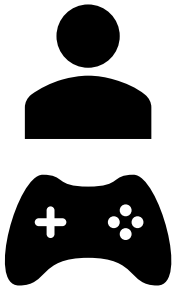
Output window

3D file format

2D image textures

.exe

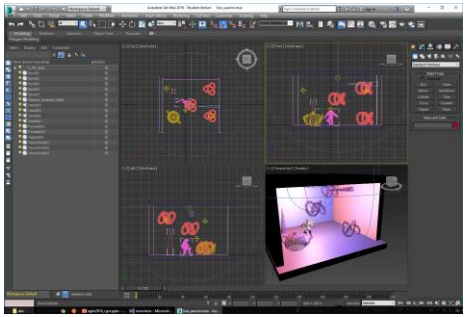
Application



User input

Plugin

API



AUTODESK 3DS MAX

Graphics engine

OpenGL

freeglut

glm

...

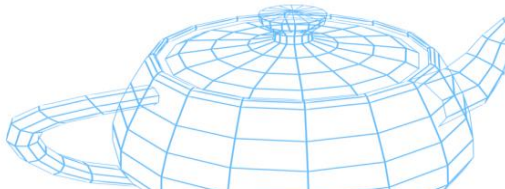
Get 3D Studio Max

- Register an account using your @SUPSI email on this page:

  https://www.autodesk.com/education/free-software/3ds-max

- Download and install 3D Studio Max **2022** (stick to version 2022 even if a newer version exists).

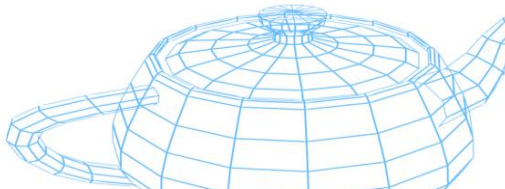
- Only available for Windows:
  - Virtualization “works” but is unstable.
  - If you really cannot access a native Windows machine within your group, talk to the teacher.



3D graphics engine

- A 3D real-time graphics engine is usually a library or SDK that provides a higher abstraction layer on top of some lower-level graphics APIs (OpenGL, DirectX, Vulkan, …):
  - It allows developers to work in terms of objects, materials, light sources rather than passing vertices, computing normal vectors, initializing contexts, allocating buffers, etc.

- 3D graphics engines expose their functions through an API:
  - Famous 3D engines have a full-fledged SDK often including visual editors, like Unity, Unreal Engine, CryEngine, OpenSceneGraph, JMonkey, etc.:
    - In addition, most engines include a physics engine, positional audio, level editors, AI and are more generally referred to as **game engines**.



3D graphics engine examples

- Common features:
  - Multi-platform (Win/MacOS/Linux) and cross-device (PC/console/mobile) rendering:
    - Using different APIs (OpenGL, DirectX, WebGL, OpenGL|ES, …).
  - Corollary tools (level editors, importers, converters, …).
  - Different licensing agreements available.
  - Integrated physics, audio, and animation engines.
  - Scripting, visual editors.

Commercial	Free/open source
Unreal Engine (www.unrealengine.com)	OGRE (www.ogre3d.org)
CryEngine (www.cryengine.com)	Irrlicht (irrlicht.sourceforge.net)
Unigine (www.unigine.com)	Minko (www.minko.io)
Unity Engine (www.unity3d.com)	MVisio (www.peternier.com)

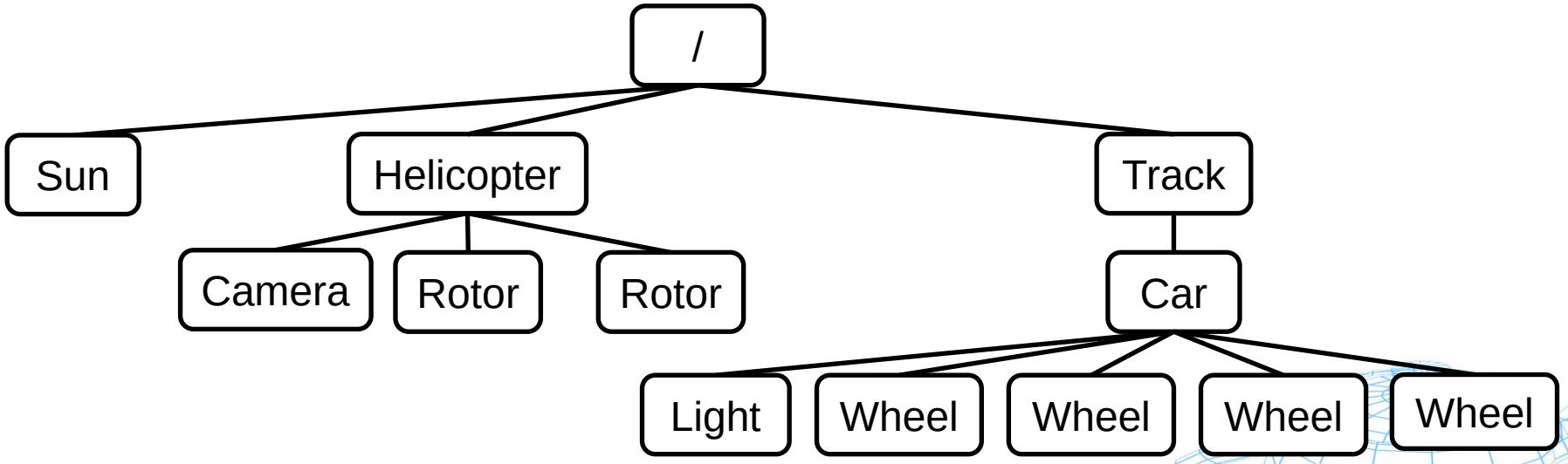
API example (MVisio)

```
#include <mvisio.h>

int main(int argc, int argv[])
{
   // Initialize the graphics engine:
   MVisio::init();

   // Load full scene graph (textures, lights, models, etc.):
   MVNode *scene = MVisio::load("my3DScene.mve");

   // Display the scene:
   MVisio::clear();
   MVisio::begin3D(scene->getMainCamera());
      scene->pass();
   MVisio::end3D();
   MVisio::swap();

   // Release resources:
   MVisio::free();
}
```

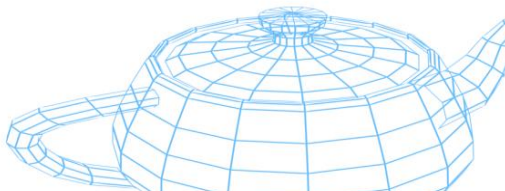
3D graphics engine

- Graphics engines organize 3D scenes into a hierarchical tree called **scene graph:**
  - Relationships between objects are expressed through parent/child dependencies using a graph.

- Each node represents one of the objects used in the scene.

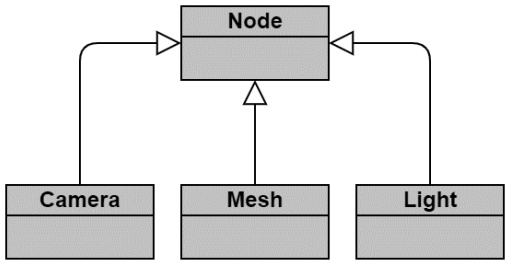


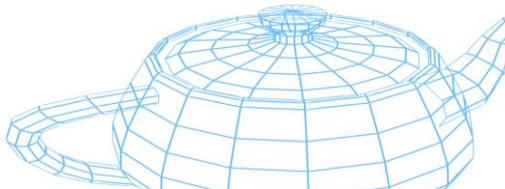
Scene graph?



Scene graph

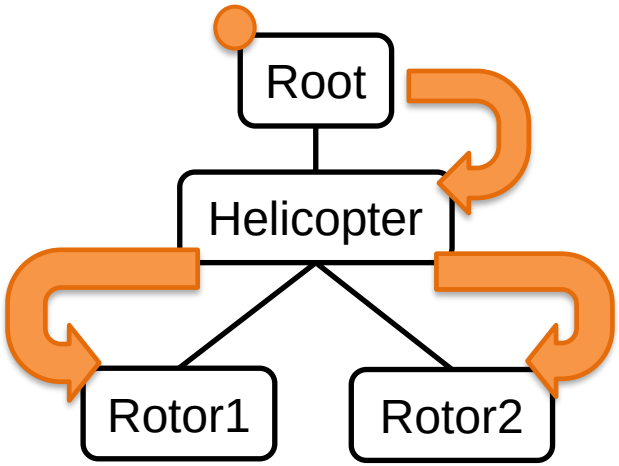
- Each element in the scene graph is derived from the same node class:
  - Node class typical methods:
    - 3D positioning methods, e.g.:
      - Set/get node matrix.
      - A way to get the final world matrix.
      - Commodity methods for basic transformations.
    - Hierarchical tree management:
      - Set parent node, add child node, remove child node, ...
      - Get parent node, get number of children, get child, …
      - Usage of **`std::vector`** or **`std::list`** recommended.





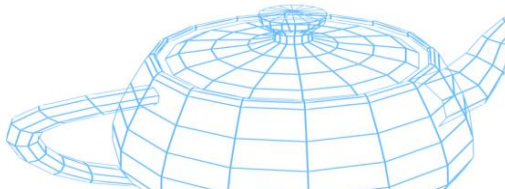
Scene graph

- By parsing the scene graph, the software can determine the final position of each object and call its rendering method passing the resulting matrix as argument:



Object	Final matrix	Rendering method
Root	M = Root	root.render(C^{-1} * M);
Helicopter	M = Helicopter * Root	helicopter.render(C^{-1} * M);
Rotor1	M = Rotor1 * Helicopter * Root	rotor1.render(C^{-1} * M);
Rotor2	M = Rotor2 * Helicopter * Root	rotor2.render(C^{-1} * M);

C = final camera matrix



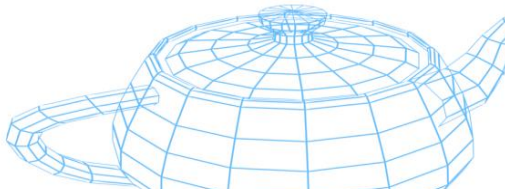
3D graphics engine – main components

Object

Base class used by all the derived classes. This class is responsible for keeping track of the existing objects, forcing some required API (virtual) methods (e.g., `render()`), and providing a unique ID to each object.

Node

Extends the previous Object class with the required functions to locate the object in the 3D space (through a matrix) and in a hierarchy (through a hierarchical structure). Should also implement a function to quickly get a given node's matrix in world coordinates.



3D graphics engine – main components

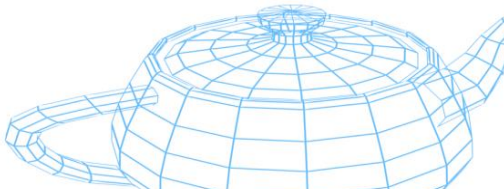
Camera

This class represents a camera. Settings should comprise both orthographic and perspective projections, and the necessary math to retrieve the camera inverse matrix.

Mesh

Class responsible for storing a single 3D object (including its vertices, texturing coordinates, and a reference to the used material). The class includes the necessary methods for passing data to OpenGL.
For now, just render 3D cubes.

More to come later…



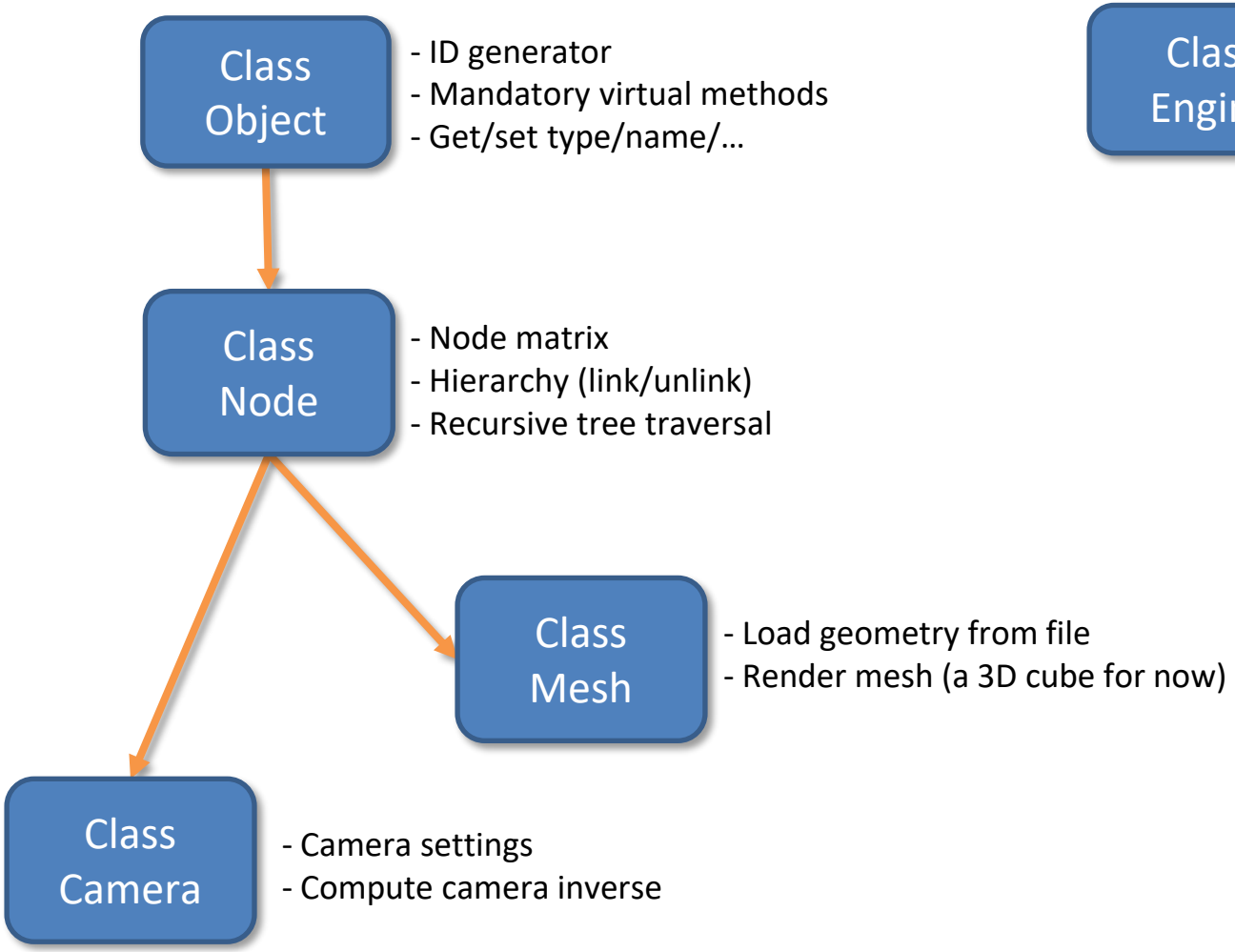
3D graphics engine – main components

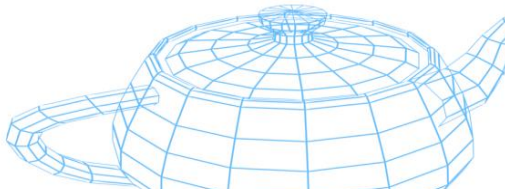
Engine

The engine class is the main component of the API. It's a single class (either static or singleton) responsible for initializing and interacting with the OpenGL context and the various engine components.

```
#include <mvisio.h>

int main(int argc, int argv[])
{
   // Initialize the graphics engine:
   MVisio::init();

   // Load full scene graph (textures, lights, models, etc.):
   MVNode *scene = MVisio::load("my3DScene.mve");

   // Display the scene:
   MVisio::clear();
   MVisio::begin3D(scene->getMainCamera());
      scene->pass();
   MVisio::end3D();
   MVisio::swap();

   // Release resources:
   MVisio::free();
}
```

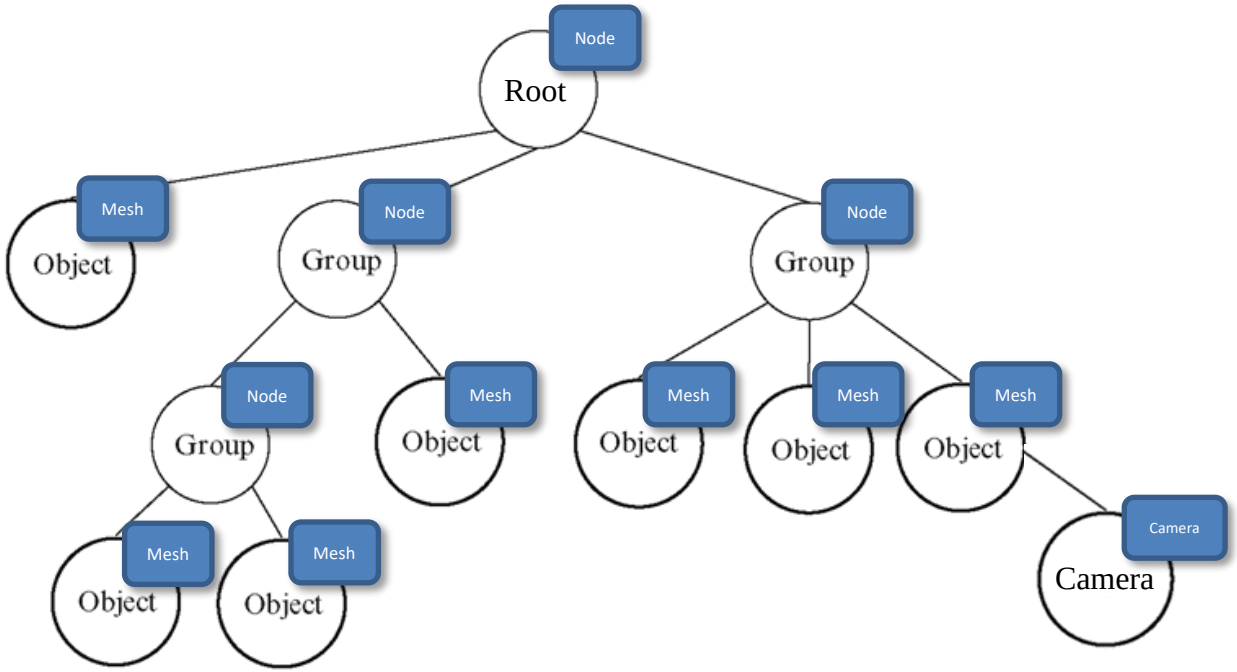
3D graphics engine – main components

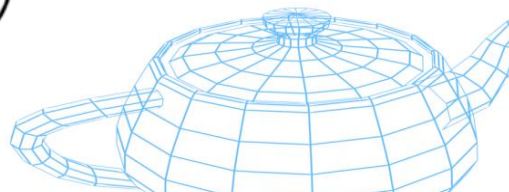




Scene graph

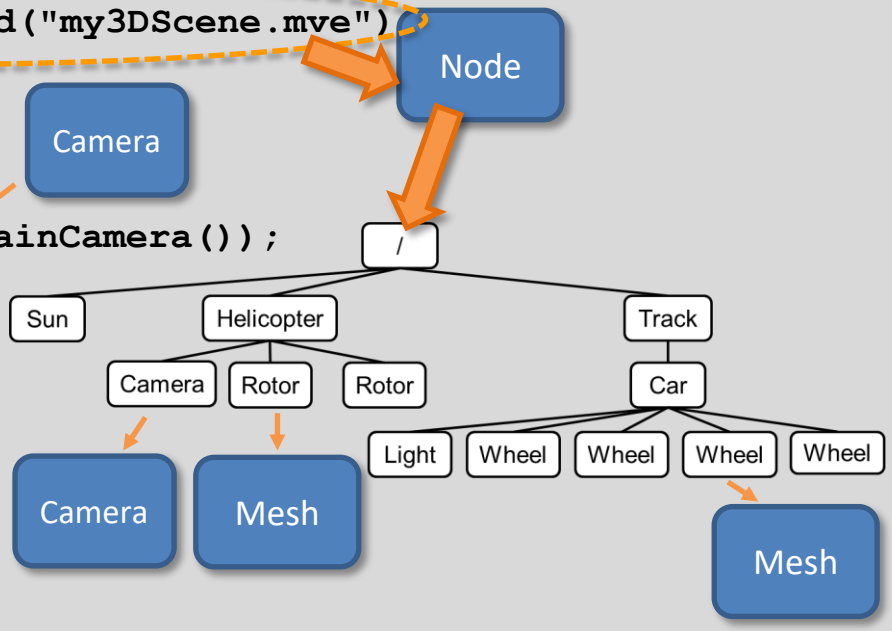
- Typical scene graph elements:
  - Meshes, light sources, etc.
  - Auxiliary classes such as helpers, groups, etc.





3D graphics engine – main components

```
#include <mvisio.h>

int main(int argc, int argv[])
{
   // Initialize the graphics engine:
   MVisio::init();

   // Load full scene graph (textures, lights, models, etc.):
   MVNode *scene = MVisio::load("my3DScene.mve");

   // Display the scene:
   MVisio::clear();
   MVisio::begin3D(scene->getMainCamera());
      scene->pass();
   MVisio::end3D();
   MVisio::swap();

   // Release resources:
   MVisio::free();
}
```



Implementation hints

- Decide which dependencies will be integrated in the graphics engine and which ones will be also required client-side:
  - If you put a dependency in one of your engine's .h files, that same dependency will be required client-side!
  - Use wrapping to reduce third-party dependencies:
    - Ideally, only GLM should be used client-side.
  - If needed, replicate the (few) required definitions in your engine's include files (e.g., the definition of special keys provided by FreeGlut).

- When you wrap FreeGlut, consider using the **`glutMainLoopEvent()`** method instead of **`glutMainLoop()`** to avoid losing control:
  - Also remember that you can still define callback functions client-side and forward pointers to such functions to the wrapped FreeGlut within your graphics engine library.

- If really needed, consider using opaque structures and pointers (https://en.wikipedia.org/wiki/Opaque_pointer).