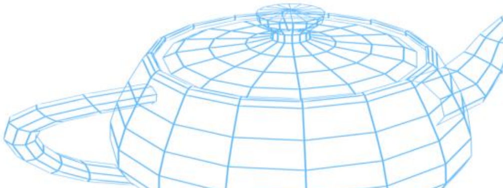
Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative

SUPSI

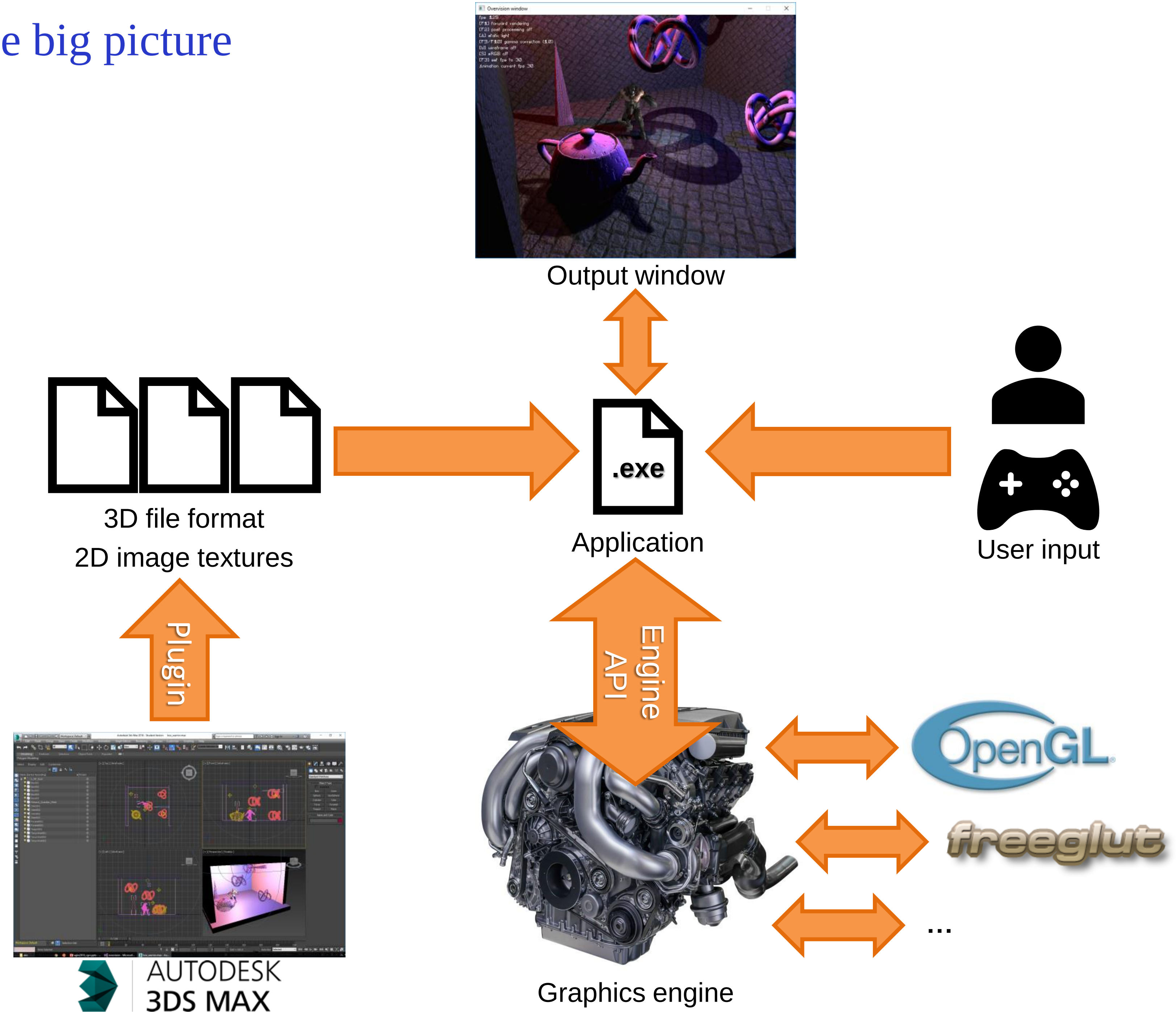
Computer Graphics

3D Graphics Engines: a few considerations

Achille Peternier, lecturer



The big picture



Overvision window

```
fps: 125
[F1] forward rendering
[F2] post processing off
[A] static light
[F9/F10] gamma correction (1.0)
[W] wireframe off
[S] sRGB off
[F3] set fps to 30
Animation current fps 30
```

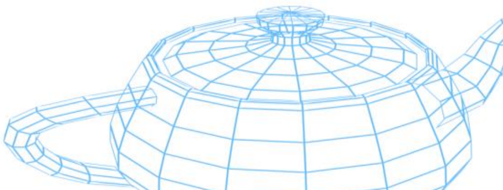
Get 3D Studio Max

- 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 **2018** (stick to version 2018 even if a newer version exists).

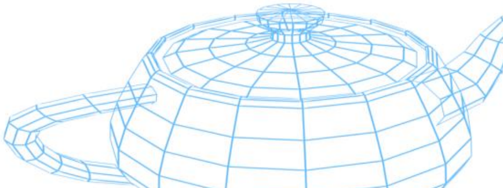
- Available for Windows only.



3D graphics engine

- A 3D real-time graphics engine is usually a library 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	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("bar.mve");

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

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

API example (Ogre3D)

```
#include <Ogre.h>

int main(int argc, char *argv[])
{
   Ogre::Root *root = new Ogre::Root("", "");

   // Load the rendersystem:
   root->loadPlugin("RenderSystem_GL");
   root->setRenderSystem(*(root->getAvailableRenderers().begin()));
   root->initialise(false);

   Ogre::RenderWindow *window = root->createRenderWindow("Hello World!",
                                                         800, 600, false);
   window->setActive(true);
   window->setAutoUpdated(true);
   window->setDeactivateOnFocusChange(false);

   Ogre::SceneManager *sceneMgr = root->createSceneManager(Ogre::ST_GENERIC);
   ...
```

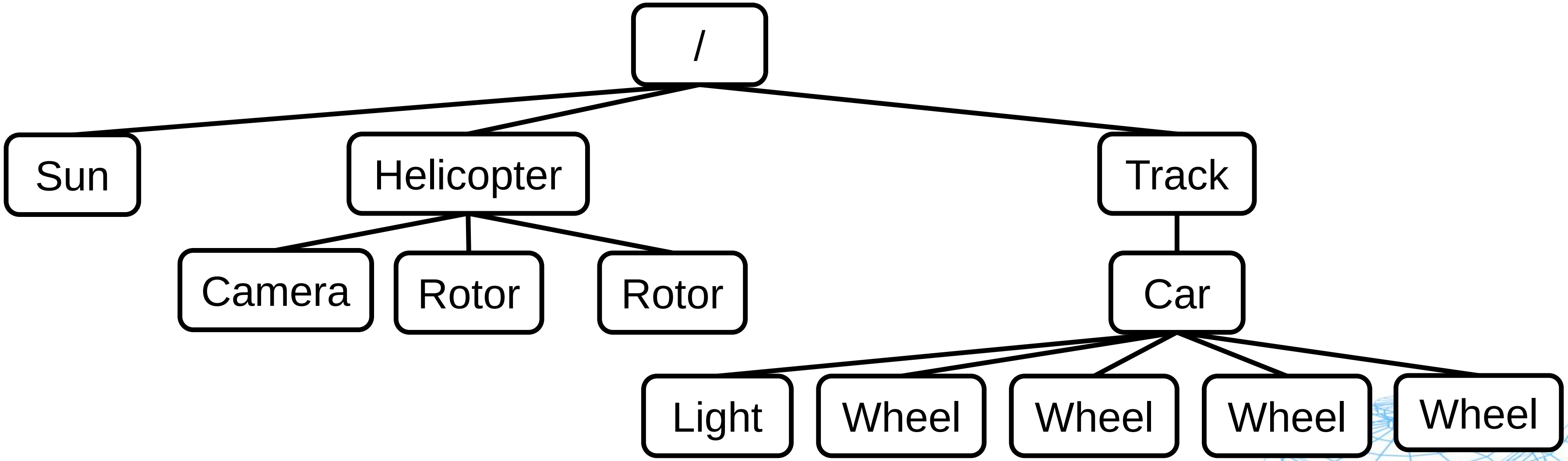
API example (Ogre3D)

```
   ...
   // Viewport and camera:
   Ogre::Camera *camera = sceneMgr->createCamera("cam");
   Ogre::Viewport *viewport = window->addViewport(camera);
   viewport->setClearEveryFrame(true);

   // TODO: set-up your camera, resources, lighting, objects...

   while (true) {
      // TODO: do your game logic here

      Ogre::WindowEventUtilities::messagePump();
      if (!root->renderOneFrame())
         break;
   }

   // Clean up:
   delete root;
   return 0;
}
```

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.



/

Sun

Helicopter

Track

Camera

Rotor

Rotor

Car

Light

Wheel

Wheel

Wheel

Wheel

3D graphics engine – main components

Engine

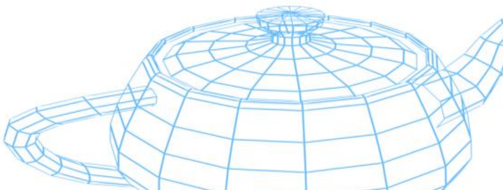
The engine class is the main component of the API. It’s a single class (either static or singleton) responsible for initializing the OpenGL context and main modules.

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) ethods and providing a unique ID to each object.

Node

Extends the 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).



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.

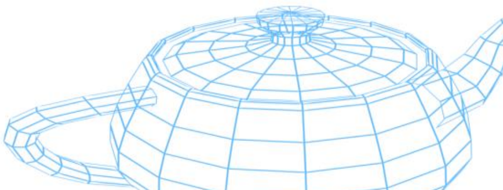
Light

Light class that implements the main types of light introduced in the course. This class includes the necessary methods for applying its settings to OpenGL. *(More about in the OpenGL 2 chapter)*

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 transferring data to OpenGL.

(More about in the OpenGL 3 chapter)



3D graphics engine – main components

List

Contains a list of instances, each one with its own properties (such as position, material, etc.). Matrices are stored in world coordinates after being evaluated according to their hierarchy.

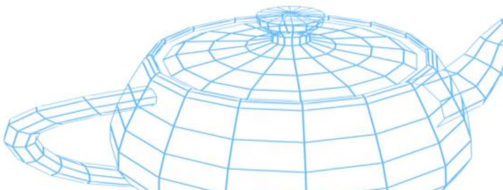
Material

Contains all the parameters necessary to define a material. It enables to change material properties and it is responsible for transferring its settings to OpenGL through the necessary methods.

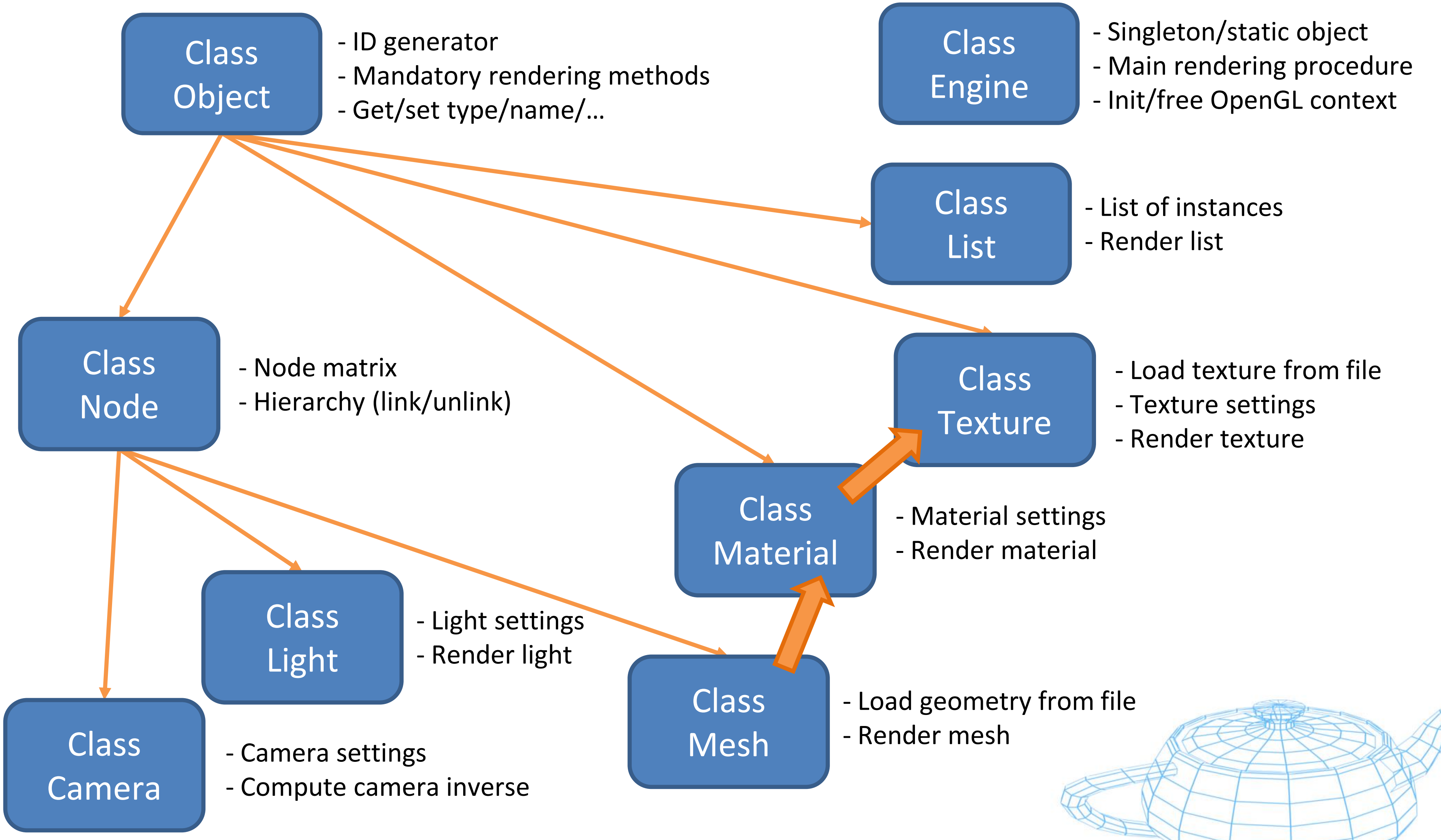
(More about in the OpenGL 2 chapter)

Texture

This class represents a texture. It is responsible for loading data from a file into an OpenGL texture and for passing its settings to the OpenGL API. *(More about in the OpenGL 3 chapter)*



3D graphics engine – main components



Class Object

- ID generator
- Mandatory rendering methods
- Get/set type/name/…

Class Engine

- Singleton/static object
- Main rendering procedure
- Init/free OpenGL context

Class List

- List of instances
- Render list

Class Node

- Node matrix
- Hierarchy (link/unlink)

Class Texture

- Load texture from file
- Texture settings
- Render texture

Class Material

- Material settings
- Render material

Class Light

- Light settings
- Render light

Class Mesh

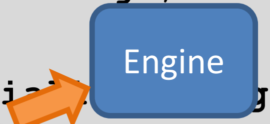
- Load geometry from file
- Render mesh

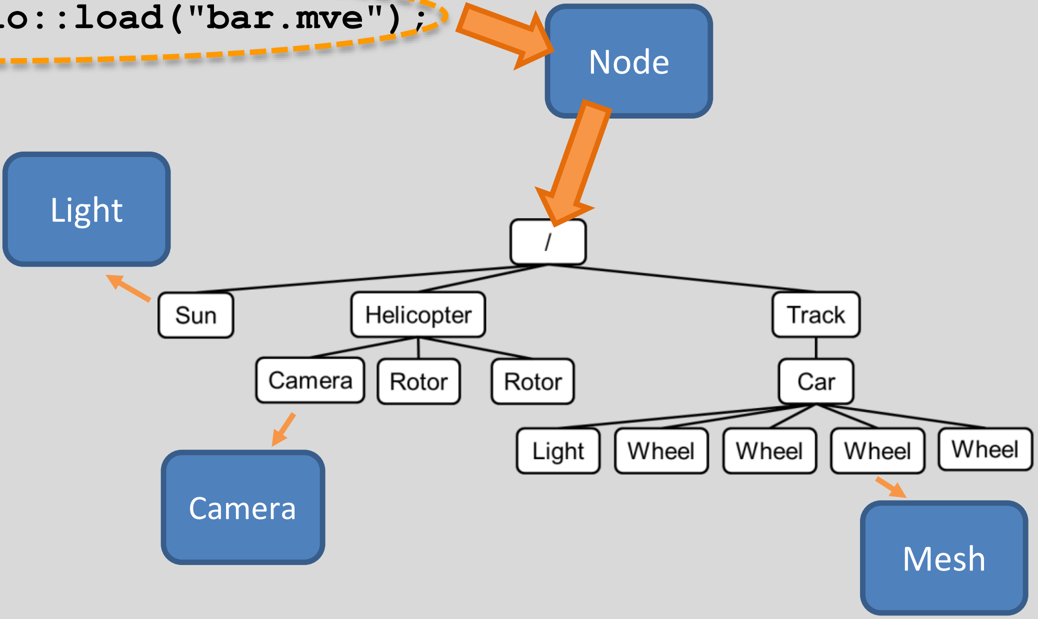
Class Camera

- Camera settings
- Compute camera inverse

3D graphics engine – main components

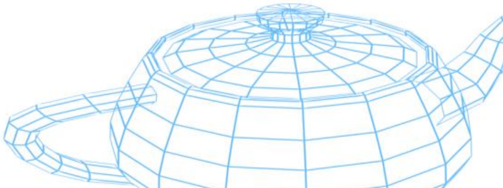
```
#include <mvisio.h>

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

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

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

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





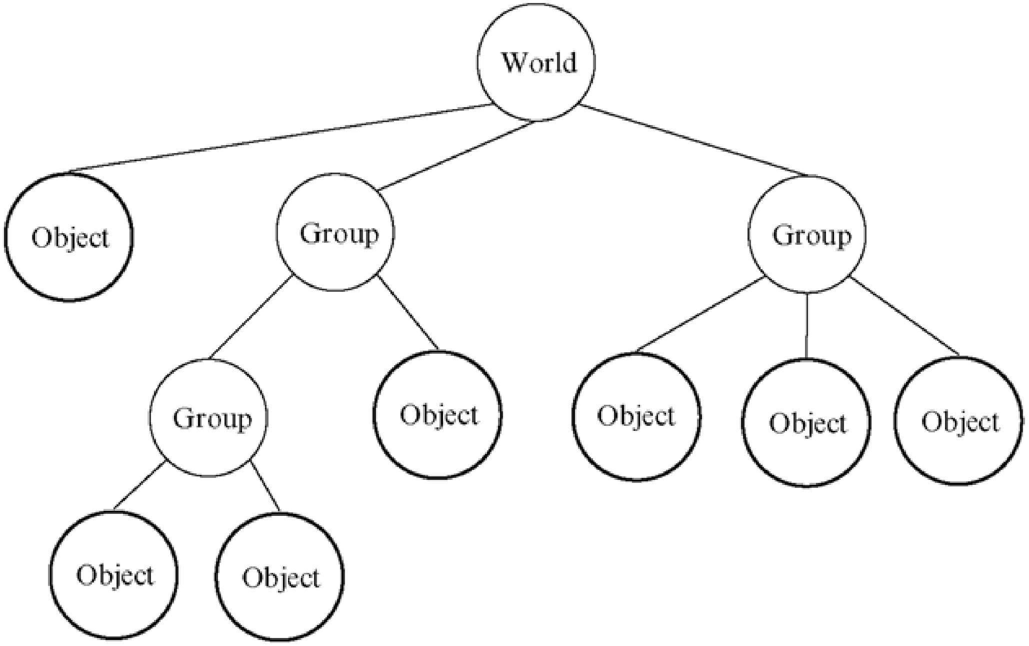
Scene graph

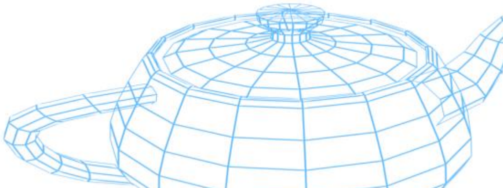
- Each element in the scene graph is derived from the same node class:
  - Node class typical methods:
    - 3D positioning methods, e.g.:
      - Set/get node matrix.
      - A way to get the final world matrix.
      - Commodity methods for basic transformations.
    - Hierarchical tree management: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



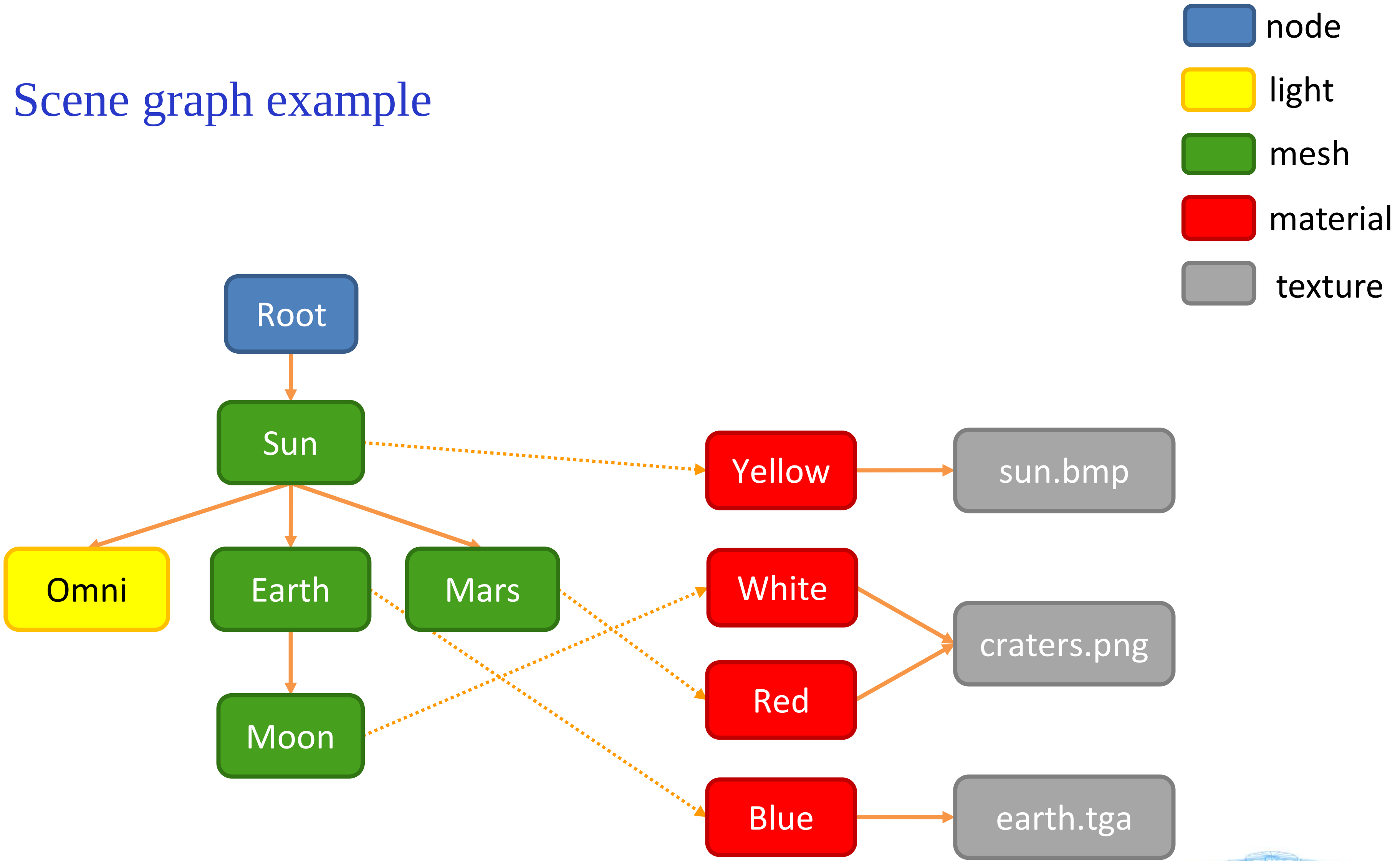
Scene graph

- Typical scene graph elements:
  - Light sources, meshes, etc.
  - Auxiliary classes such as helpers, groups, etc.
  - Materials, textures, etc. are used by meshes but are not directly part of the scene graph.





Scene graph example



🟦 node

🟨 light

🟩 mesh

🟥 material

⬜ texture

Root

Sun

Omni

Earth

Mars

Moon

Yellow

White

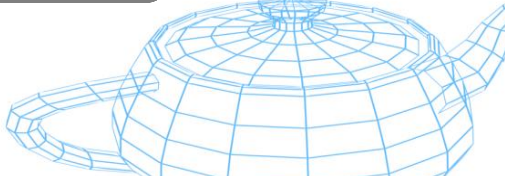
Red

Blue

sun.bmp

craters.png

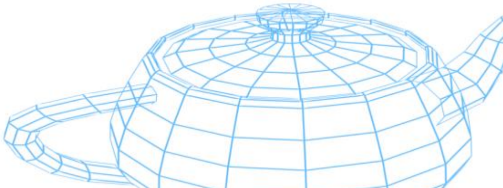
earth.tga



Instancing

- One same element (mesh, light, etc.) can be rendered multiple times at different coordinates and/or using different parameters.

- Instead of directly rendering an element, you can store a **list** of objects with specific properties (e.g., using a different matrix and material each time):
  - The list is parsed and each entry is rendered using the parameters stored in the list:
    - The list can also be sorted to render light sources first, then meshes.
  - In addition, the list can be rendered from a specific point of view by passing a camera:
    - Each matrix in the list is multiplied by the inverse of the camera matrix.
    - You can re-render the same scene from different points of view without refreshing the list's entries.



Instancing

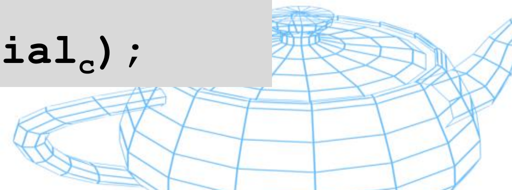
Code example:

```
planet.setMatrix(matrixA);
planet.setMaterial(blue);
list.pass(planet);

planet.setMatrix(matrixB);
planet.setMaterial(red);
list.pass(planet);

planet.setMatrix(matrixC);
planet.setMaterial(blue);
list.pass(planet);

Engine.render(camera, list);
```

List of objects to render:

Mesh	Position	Material
Planet	Matrix A	Blue
Planet	Matrix B	Red
Planet	Matrix C	Blue

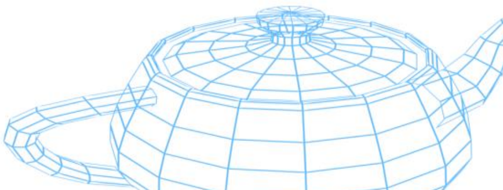
```
for c = each element of the list
   Mesh$_c$->render(camera$^{-1}$ * Position$_c$, Material$_c$);
```



Instancing

- The **`pass()`** method should be recursive and parse all the child nodes linked to the parent node:
  - At each recursion, invoke the **`pass()`** method of the child node:
    - The child node matrix is multiplied by the parent node matrix.

- In this way, invoking the **`pass()`** method on the root node will fill the list with the content of the entire scene, evaluated according to its position and hierarchical structure:
  - The list will contain all the scene objects in world coordinates.
  - Multiple scene graphs (or the same processed multiple times) can be rendered into the same list.
  - Multiple lists can be used (e.g., one for 2D and one for 3D rendering).



Scene graph?

Implementation hints

- Decide which dependencies will be integrated in the graphics engine and which ones will be required also 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).