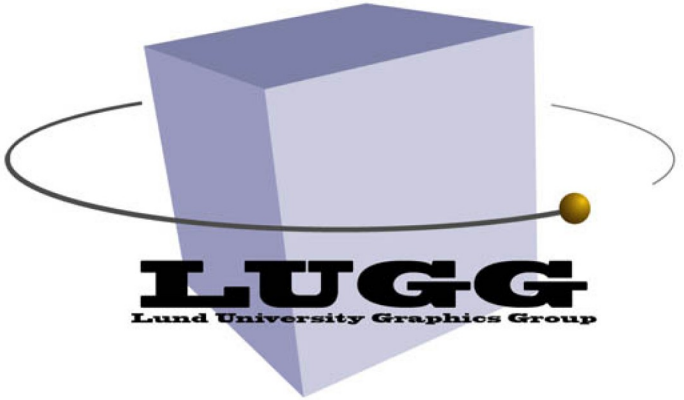
Seminar:
Assignment 1

Magnus Andersson (magnusa@cs.lth.se)



This seminar

A lot of ground to cover...

- Assignment
- C++ crash course
- RenderChimp

Assignment 1

1. Getting started
2. Building a Scene Graph
3. Simple Game
4. Playing around in Shaders

Assignment 1

1. Getting started
   - Download Visual Studio Express
   - Open RenderChimp.sln
   - Compile and Run (should get a warning)

   (if there is time, VS demo)

Assignment 1

2. Building a Scene Graph
   - World node
   - Creating and managing resources and nodes
     - Create your own object using a VertexArray and an IndexArray
   - Hierarchical transformation

Assignment 1

3. Simple Game
   - Transforming objects over time
   - Input

   (if there is time, show demo)

Assignment 1

4. Playing around in Shaders
   - This is all the *shader* you'll ever have to see...
   - Getting a value all the way from the platform to the shader
   - Time dependent effect
     - Just one or a few lines of code...

Assignment 1

Errors and warnings end up in:

log.txt

Memory information ends up in:

mem.txt

Class Definition

In point.h:

```
class Point
{
      public:
            Point(float x, float y); // constructor
            float getX(void) const;  // accessor: const means does not change
                                           state of object
            float getY(void) const;

      protected:
            float mX;    // member attributes
            float mY;
};
```

The actual implementation is in point.cpp

Implementation of Point

In point.cpp:

```
Point::Point(float x,float y) // Point:: indicates which class
{
	mX=x;
	mY=y;
}

float Point::getX(void) const // getY() in the same way
{
	return mX;
}
```

Declarations and Definitions

Function declaration:

In header file (for example):

```
bool finished(int t);  // note semi-colon instead of function body
```

In cpp-file, function definition:

```
bool finished(int t)
{
	if(t>1) return true;
	else return false;
}
```

Allocation

```
void func(void)                    // function not belonging to class (no ::)
{
      int a;
      a=sin(0.314);
      Point pl;
      Point *p = new Point(10.0, 20.0);      // a pointer
      Point *pa = new Point[20];             // array of 20 point objects
}
```

When func() is entered, a & pl are allocated on the stack, and when exited, a & pl are automatically deleted.

p and pa is allocated using new, which means that you need to delete it at some point: delete p; delete [] pa;

There is **NO garbage collection** in C++.

Destructor

Used when a class allocates memory using new.

The destructor deletes what it has allocated

```
class Point
{
	public:
	Point();
	~Point();		// destructor
};
```

In point.cpp:

```
Point::~Point()
{
	// delete memory here, for example:
	delete mNameOfPointString;   //if there was such a variable
}
```

Inheritance

```
class Point
{
	public:
		virtual void update(void);
};
```

```
class TimePoint : public Point   // inherit from Point
{
	public:
		void update(void);       // overloads Point::update
};
```

Namespaces

Similar to packages in Java.

In header-file, rc.h:

```
namespace rc
{
	class Point {...};
}
```

In cpp.file:

```
#include “rc.h”
namespace rc
{
	Point::getX(void) const { return mX; }
}
```

Using namespaces

```
#include “rc.h”

void test(void)
{
      using rc::Point;
      Point p;
}

// can “import” everything from a namespace by

using namespace rc;
```

Reference and Pointer Parameters

Default as in Java: parameter is copied

Then we have pointers and references as well

```
int func(Point &pr, Point *pp)          // & is ref, * is pointer
{
      pr.setX( pr.getX()+10 );          // note .
      pp->setX( pp->getX()+10 );        // note ->
}

void test(void)
{
      Point pr(1,1);                    // stack allocation
      Point *pp= new Point(1,1);        // heap allocation

      func(pr, pp);
}
```

Functions and Arrays

An array: int a[10]; // no a.length as in Java

Array is passed as pointer to first element:
void func(int b[], int sizeOfArray) {...}
or
void func(int *b, int sizeOfArray) {...}

Operator Overloading

Regular operators can be overloaded.

For example:

```
{
    Matrix4x4 m = Matrix4x4(...);
    Matrix4x4 n = Matrix4x4(...);

    Matrix4x4 o = m * n;                    // Matrix multiplication!
}

Matrix4x4 Matrix4x4::operator*(const Matrix4x4 &v)
{
    // Matrix multiplication implementation
}
```

(take peak inside VectorMath.h and VectorMath.cpp)

Output

```
// For example, might be good for debugging sometimes

   std::cout << "drawing..." << std::endl;

// prints “drawing...” to standard output


// You can also use

   printf(“drawing...”);

// which is more C-like
```

Preprocessor

Lines beginning with #

The pre-processor is executed prior to the compiler

```
#define X

#ifdef X
      printf(“X is defined”);
#else
      printf(“X is not defined”);
#endif
```

More info

- C++ course slides
http://www.cs.lth.se/EDA031/forelasningar.shtml

- cplusplus.com
http://www.cplusplus.com/doc/tutorial/

Moving on...

RenderChimp
Scene Graph

RenderChimp

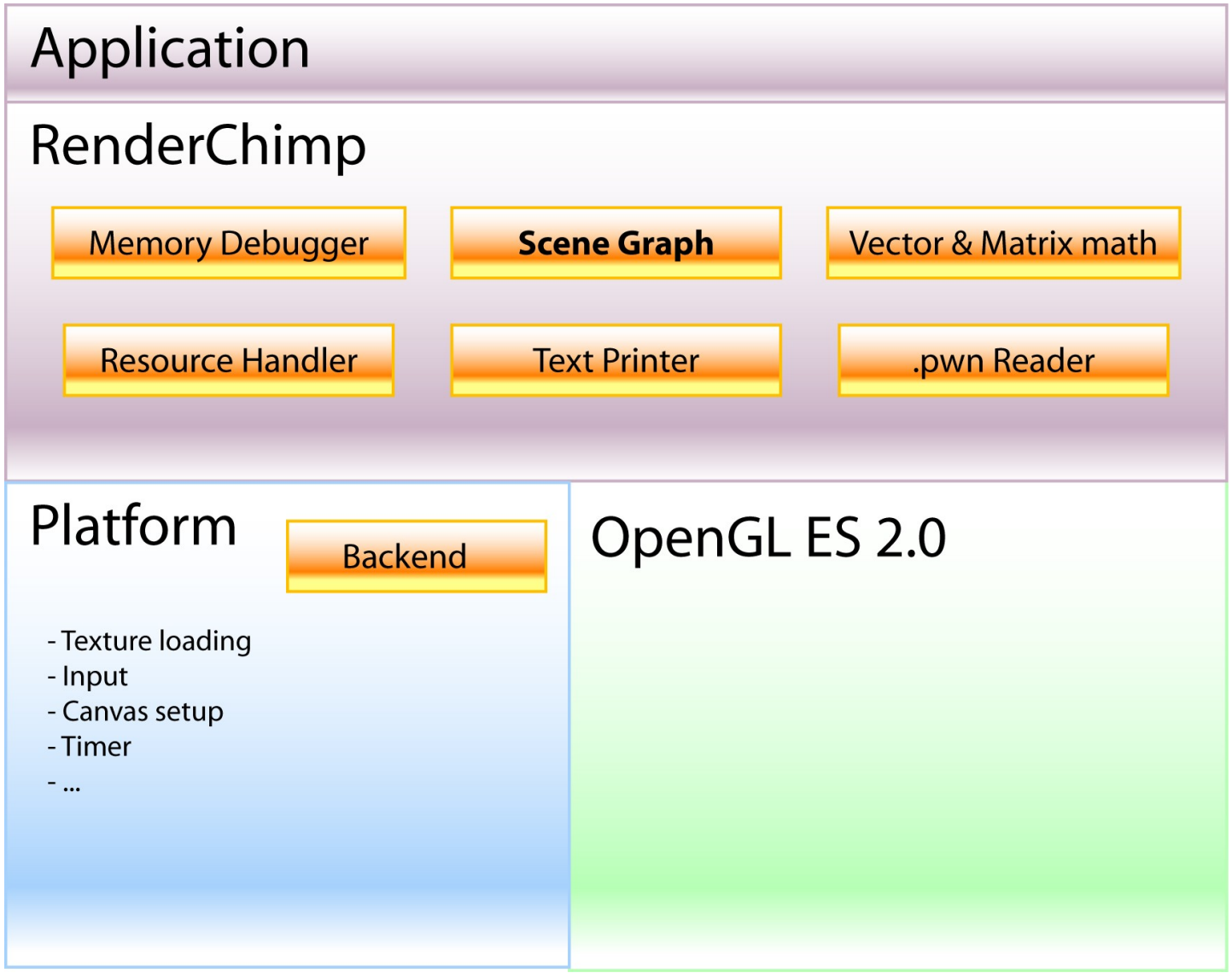
Assignment package

- Out now!
- You will need:
  - Visual Studio Express
  - The assignment package from the home page

Project package

- An updated version of the framework for use in the project will be available soon...

RenderChimp Overview

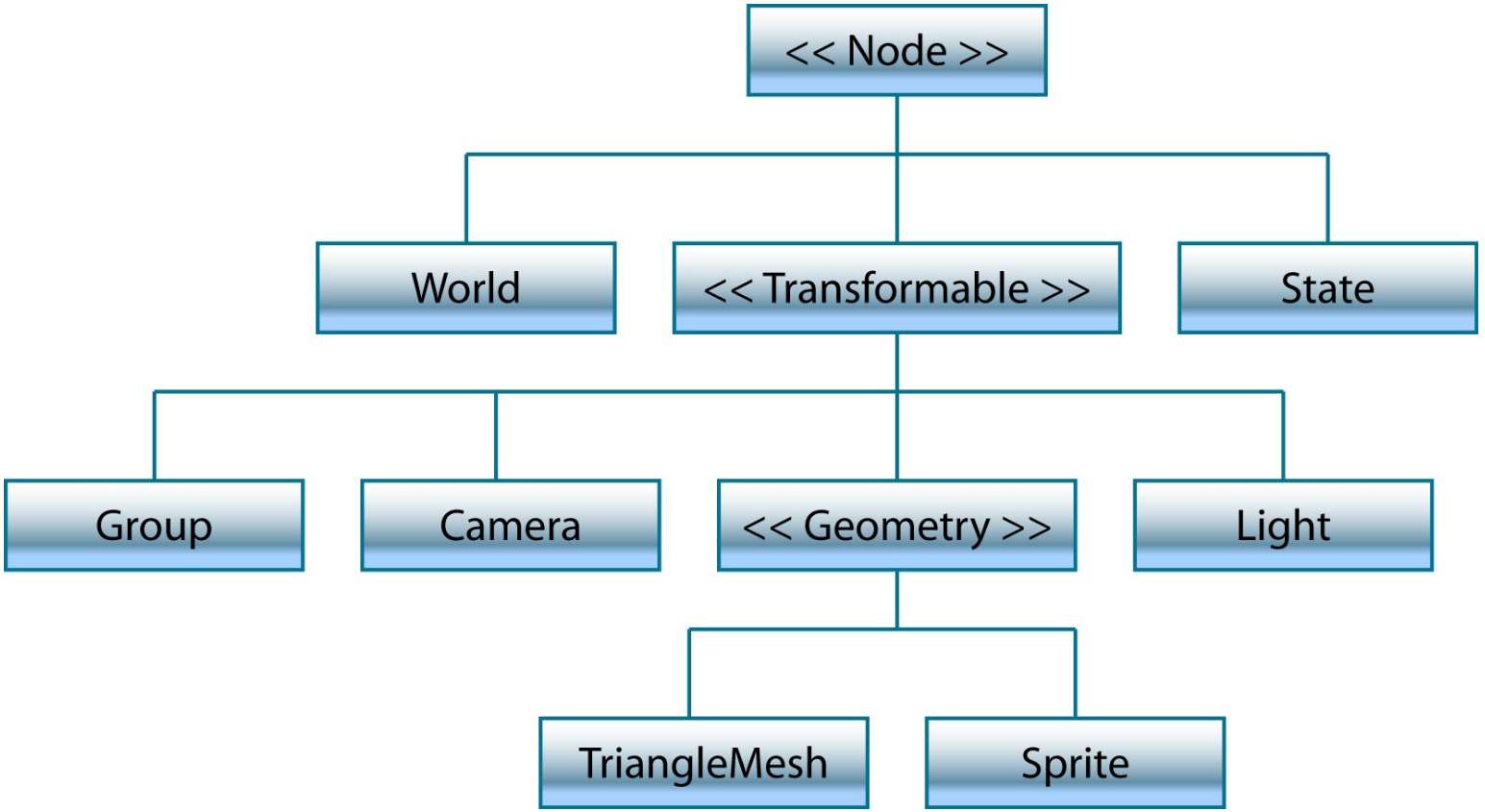


RenderChimp Overview

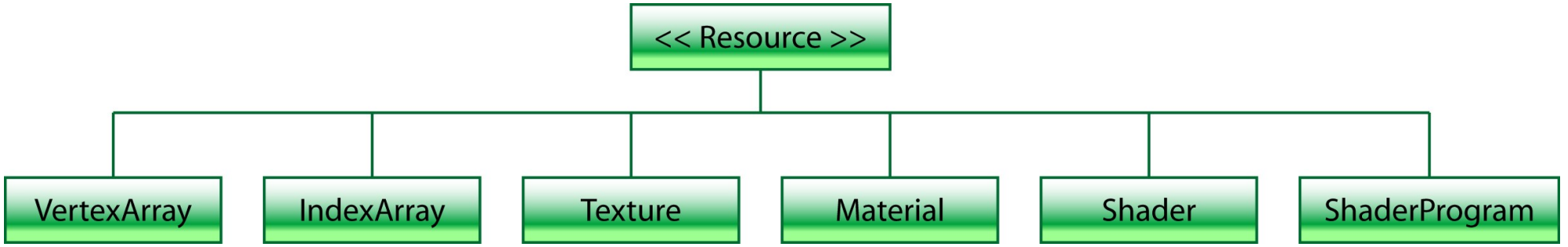
- RCInit() - called when program is loaded
- RCDestroy() - called just before program exits
- RCUpdate(DeviceState_t *ds) - called every frame

```
struct DeviceState_t {

    u32             touch;
    f32             x;
    f32             y;

    f32             roll;
    f32             pitch;
    f32             facing;

    f32             time;
    f32             timeStep;

};
```

Scene Graph **Nodes**



Scene Graph **Resources**



Nodes vs. Resources

Nodes:

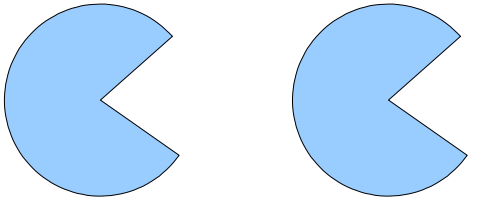
- Describe *hierarchical* relationships in a *scene*.
- Organized in a tree-like manner. One parent per node.
- *One* node describes *one* object.
- Relatively cheap (~10s - 100s of bytes)

Resources:

- Describe *data*.
- Not organized in any particular way.
- *One* resource can be instantiated *many* times.
- Relatively expensive (~10s – 1000000s of bytes)

Nodes use resources. Resources use resources. No-one uses nodes.

Nodes vs. Resources



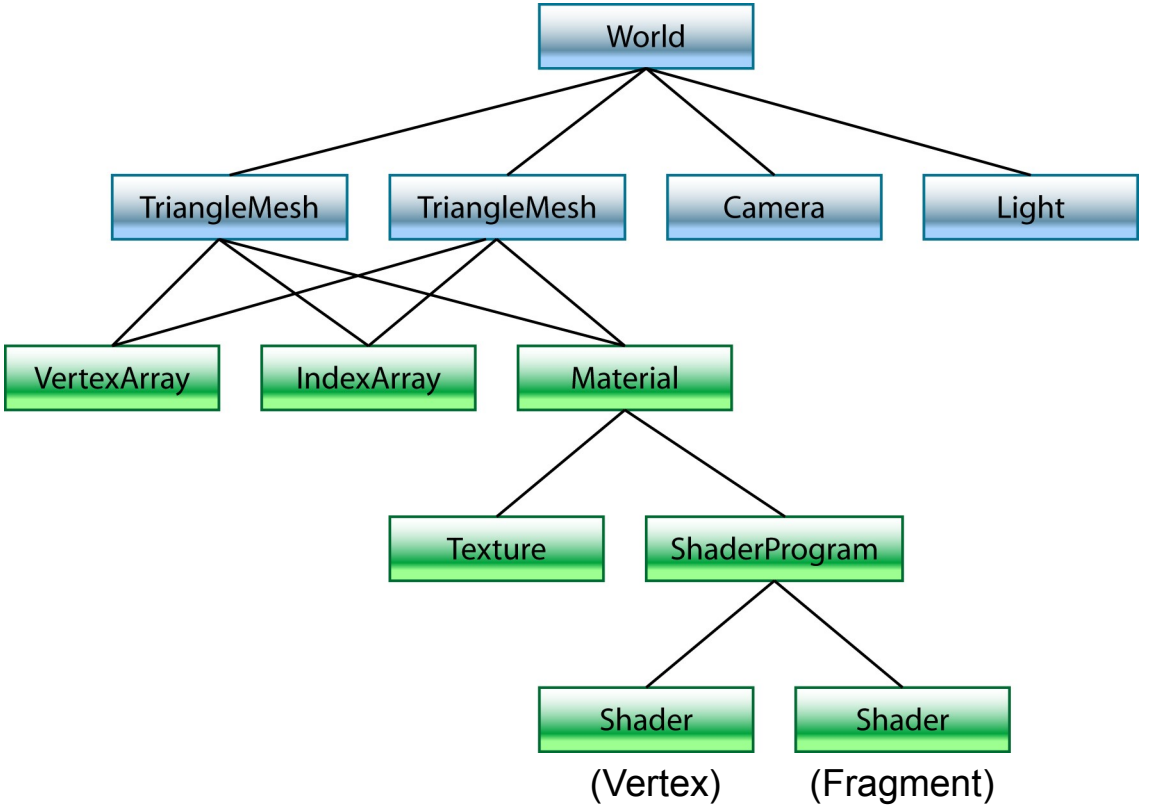
Imagine a game with two enemies...

Their triangle data is identical. Storing that data twice = twice as expensive!

...what about 50 enemies?

Solution: Shared data!

However: located at different positions in the scene.



The sceneGraph singleton

In RenderChimp there is an almighty singleton object called

sceneGraph

All creation and deletion of Nodes and Resources must go through this object. (**new** and **delete** are *not* permitted for any RenderChimp objects).

For example: To create a Sprite you may write something like this:

```
Sprite *s = sceneGraph.createSprite(“MrSprite”, “textures/smileyface.png”, ...);
```

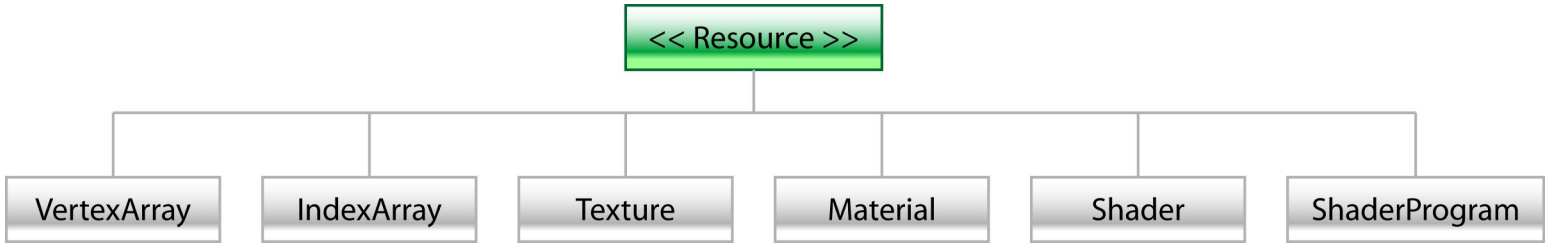
And to delete it use:

```
sceneGraph.deleteNode(s);
```

Take a look inside SceneGraph.h for more info...

Resources

Let's look through the available resource types...



- VertexArrays, IndexArrays and Materials can be cloned. This means that their data is duplicated. This is most often less expensive than re-creating your resource.

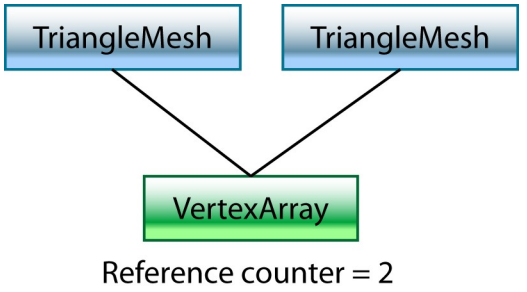
```
virtual Resource *clone(...);
```

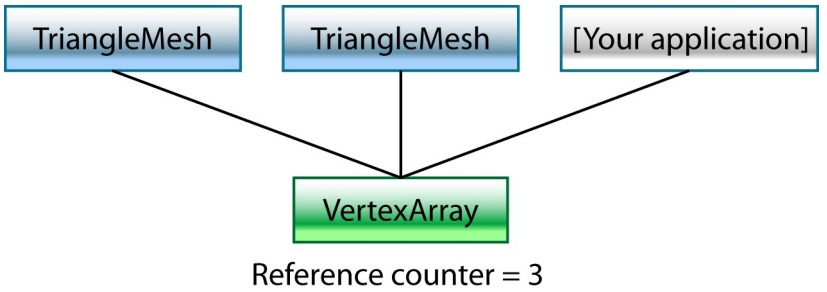
- To retrieve a new instance of a resource, use this function:

```
virtual Resource *instantiate();
```

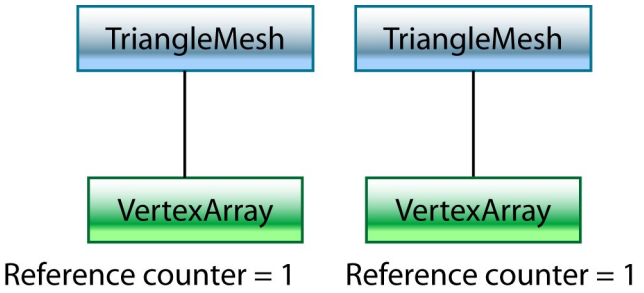
Resource instantiation and cloning

Instantiation:





Cloning:



Resource handling

Note: Resource handling differs in the Assignment package and the coming Project package...

- **For the assignment, just declare everything *permanent*, and let the sceneGraph clean up after you on shutdown.**

Resource handling

In the Project package...

Each resource is assigned a *purge level*. For instance...

Give **Menu** resources purge level 5.
Give **Player mesh** purge level 3.
Give **Level data** resources purge level 1.

To get rid of all the Level data and the Player mesh, but not the Menu, use:

```
sceneGraph.purgeResources(3);  // Purges everything <= 3
```

But, Node tree must die first! (it's your responsibility)

Resource handling

In the Project package...

Special case: purge level = 0.

Resource is associated with a *reference counter*.

- ++ every time a resource is instantiated.
- -- every time a reference to it is severed.
- If the reference counter hits 0, the resource is *deleted*.

Resource instantiation

There is a difference between... (1)

```
Sprite *s = sceneGraph.createSprite("mySpr", "textures/smoke.png", 0);
```

...and... (2)

```
Texture *t = sceneGraph.createTexture("textures/smoke.png", 0);
Sprite *s = sceneGraph.createSprite("mySpr", t, true); // (instantiate_resource =
                                                           true)
```

...and... (3)

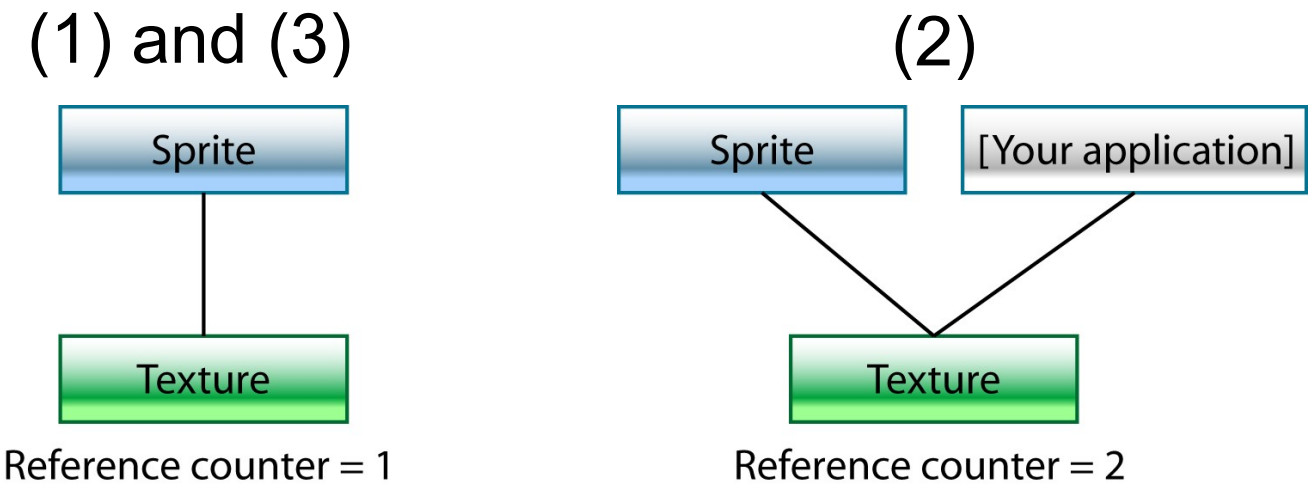
```
Texture *t = sceneGraph.createTexture("textures/smoke.png", 0);
Sprite *s = sceneGraph.createSprite("mySpr", t, false); // (instantiate_resource =
                                                            false)
```

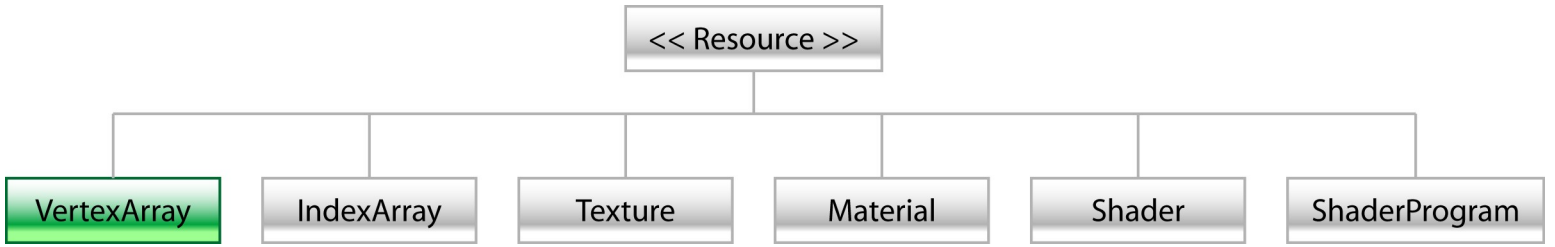
1) The texture is created within the Sprite. We don't need to manage the resource.

2) The texture is created by our application. When it's passed to the Sprite, it is instantiated. We hold one instance of the Texture (the *t pointer), as do the Sprite! This means that the resource should be deleted on *our side* at some point.

3) We created the Texture, but surrender it to the Sprite. This is equivalent to (1).

Resource instantiation





- An array of float attributes associated with each vertex.
- For example: “Vertex”, “Normal”, “Color”, “Texcoord”, ...
- Organized with attributes to the same vertex adjacent to each other.

- Set attributes using

```
void setAttribute(offset, length, name);
```

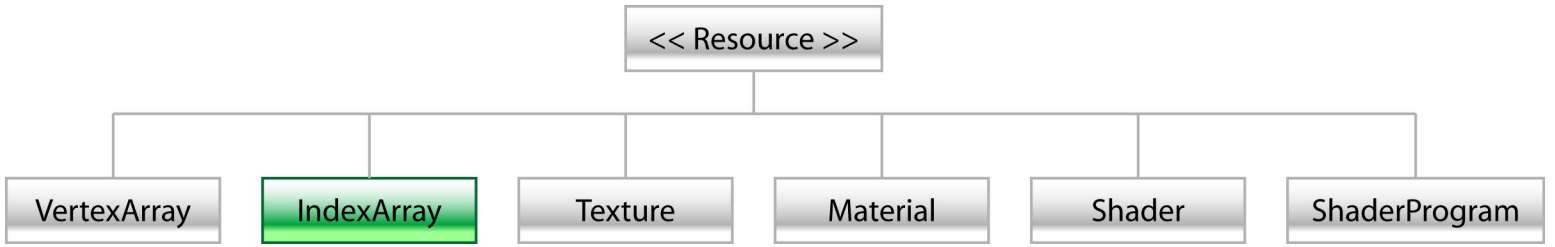
- In the example to the right we have:

```
setAttribute(0, 3, “Position”); // x, y, z floats
setAttribute(3, 4, “Color”); // r, g, b, a floats
```

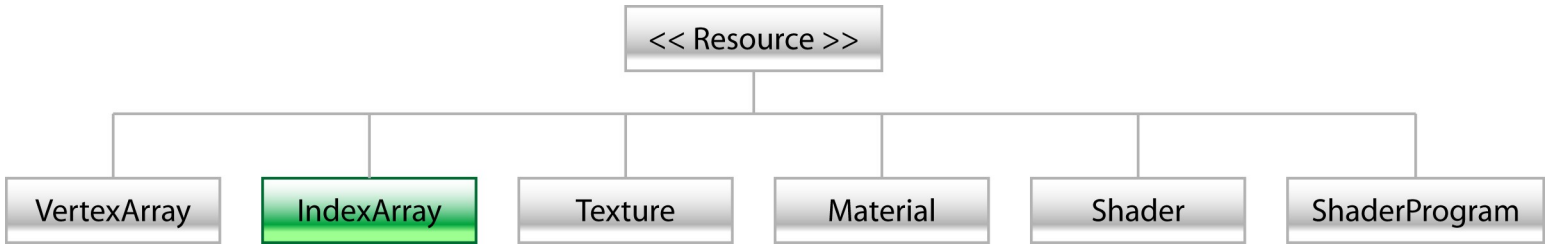
Important note: The example shaders expects the attribute name “Vertex” instead of “Position".

Vertex Array

0:	Position.x
	Position.y
	Position.z
	Color.r
	Color.g
	Color.b
	Color.a
1:	Position
	Color
2:	Position
	Color
3:	Position
	Color
4:	Position
	Color
5:	Position
	Color
	...



- Contains indices to a VertexArray
- Removes the need to repeat data in VertexArray
- TRIANGLE, TRIANGLE_STRIP, TRIANGLE_FAN



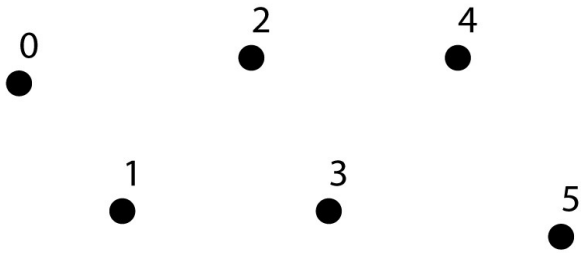
Vertex Array

0: Position.x
Position.y
Position.z
Color.r
Color.g
Color.b
Color.a
1: Position
Color
2: Position
Color
3: Position
Color
4: Position
Color
5: Position
Color
...

Index Array
(TRIANGLES)

A: 0, 1, 2
B: 1, 3, 2
C: 2, 3, 4
D: 3, 5, 4

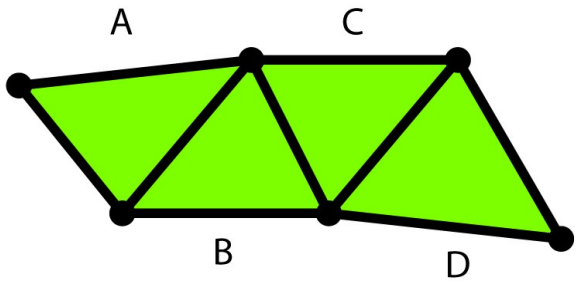
(n * 3 entries)

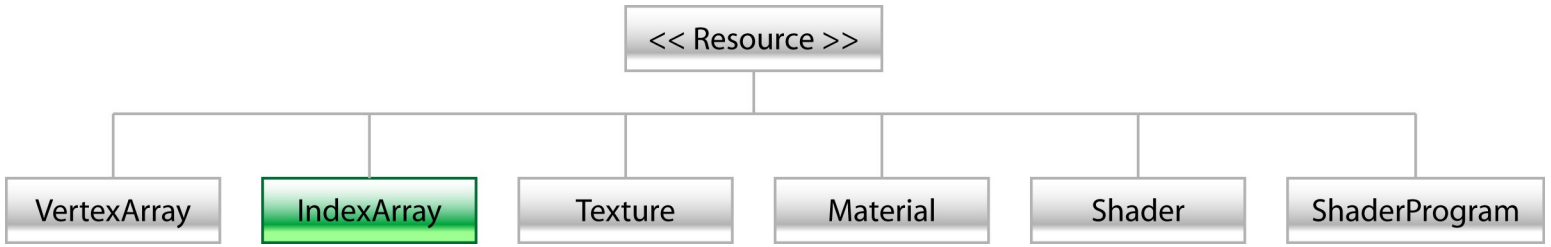


IndexArray
(TRIANGLE_STRIP)

A: 0, 1, 2
B: 3
C: 4
D: 5

(n + 2 entries)





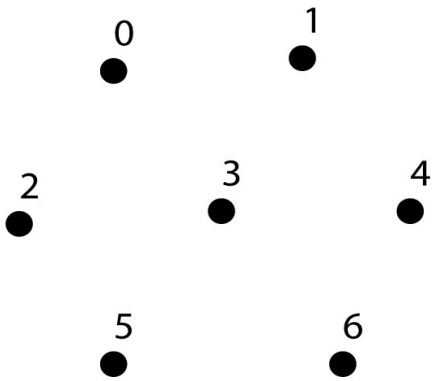
Vertex Array

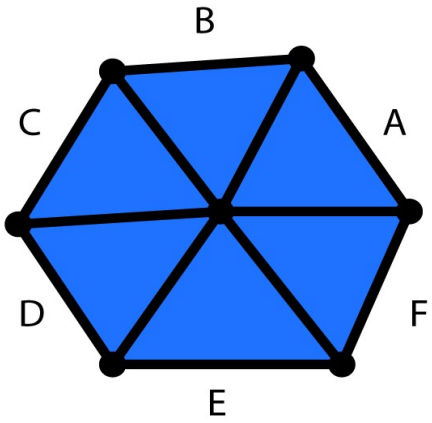
0: Position.x
Position.y
Position.z
Color.r
Color.g
Color.b
Color.a
1: Position
Color
2: Position
Color
3: Position
Color
4: Position
Color
5: Position
Color
...

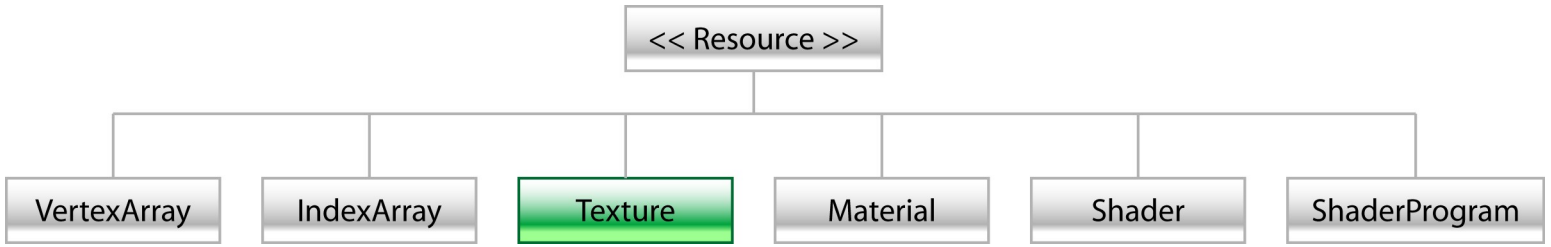
IndexArray
(TRIANGLE_FAN)

A: 3, 4, 1
B: 0
C: 2
D: 5
E: 6
F: 4

(n + 2 entries)







Basically an array of pixels.
Loaded in to texture memory.

Supported formats:

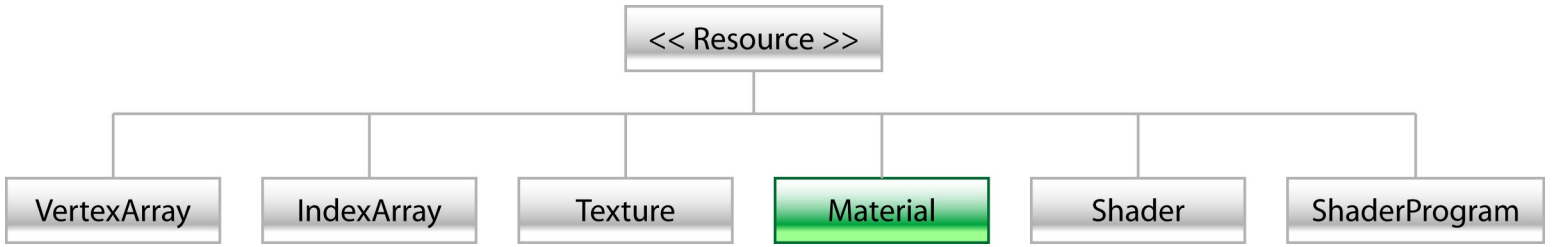
- RGBA, RGB, Grayscale

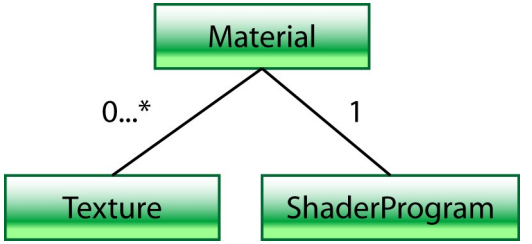
Filtering:

- Nearest neighbour, Bilinear, Trilinear

Wrapping:

- Clamp, repeat, mirrored repeat



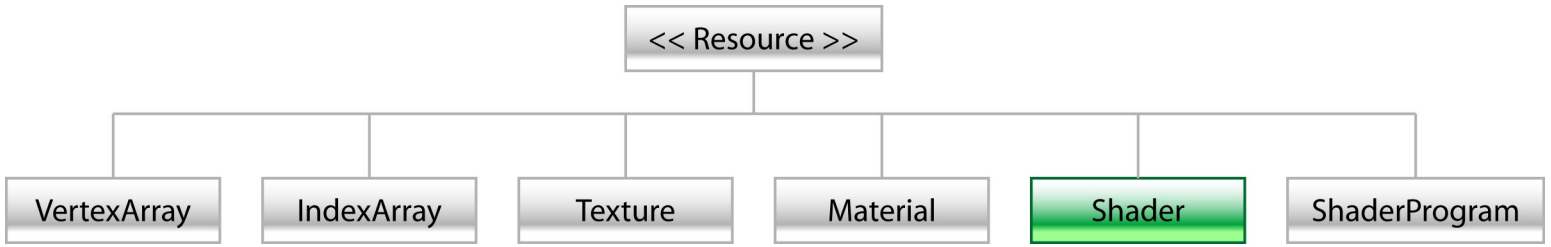


- Defines the appearance of your triangle mesh
- Essentially its purpose is to hold data for the associated ShaderProgram.

The assignment package has two example Materials:

- MaterialColorful:
  - Very simple. Interpolates color between vertices.
  - Expects “Vertex” and “Color” vertex attributes.

- MaterialPhong:
  - A textured, per-pixel phong shader.
  - Expects “Vertex”, “Normal” and “Texcoord” vertex attributes.

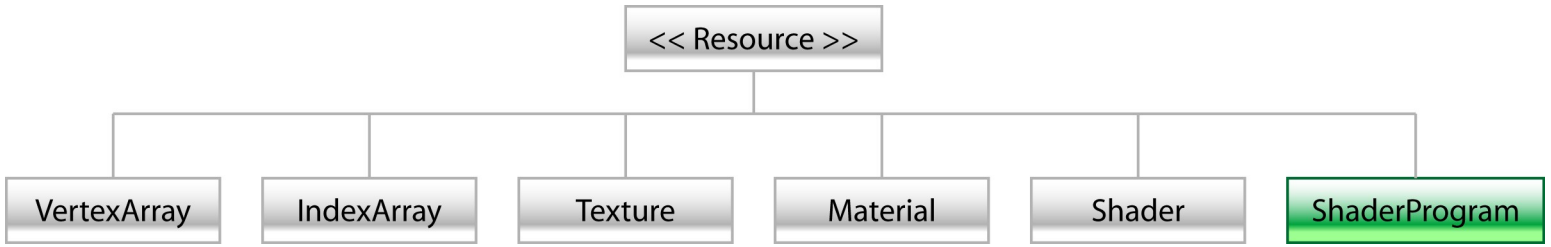


Can be either a **Vertex shader** or a **Fragment shader**

Vertex Shader
- Loaded from .vs source files.
- Per vertex calculations

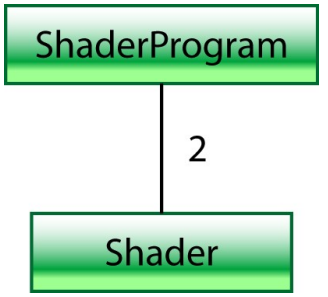
Fragment Shader
- Loaded from .fs source files.
- Per pixel (fragment) calculations

(Shaders are compiled individually at run-time)



Vertex Shader + Fragment Shader = **ShaderProgram**

- Links two compiled shaders to a program

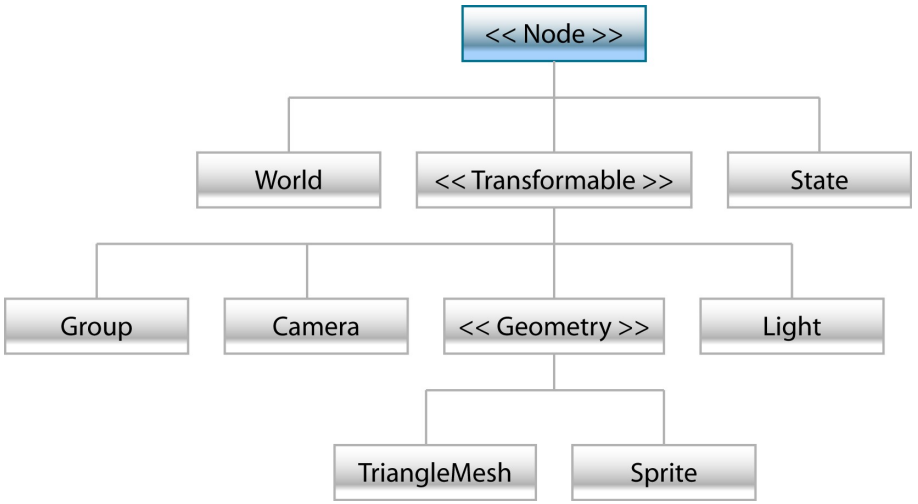


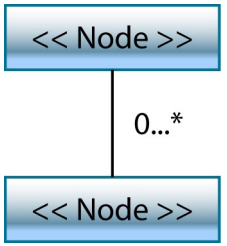
Useful for a range of different purposes:

- Materials (lighting calculations, texturing, ...)
- Per-vertex transformations (skinning, noise, ...)
- Post processing effects (depth of field, ambient occlusion, ...)
- Use your imagination!

Nodes

Let's look through the available node types...





- Each node is identified by a name:

```
void setName(...);          char *getName();
```

- Each node can have any number of children and may or may not have a parent.

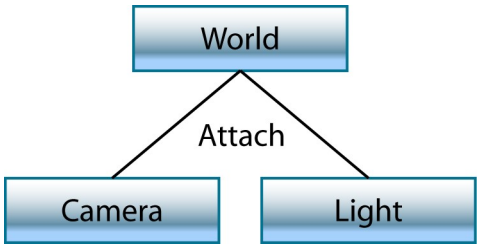
```
void attachChild(...);      void detachChild(...);
void detachFromParent(...); Node *getChild(...);
Node *getNextSibling();     Node *getParent();
```

- Duplicate the node, either by itself its entire subtree.

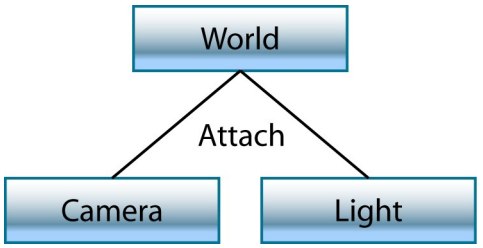
```
Node *duplicate(...);
```

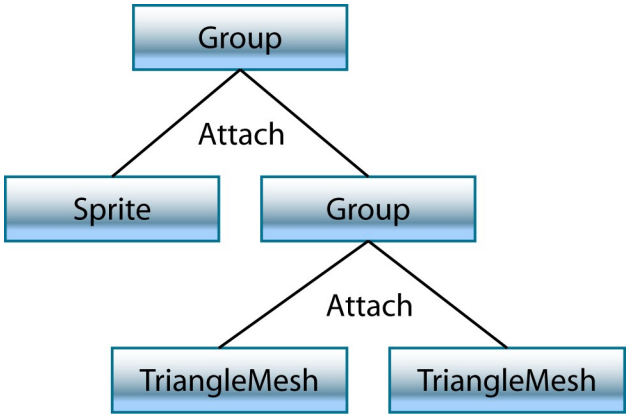
Scene graph building example

Scene graph building example

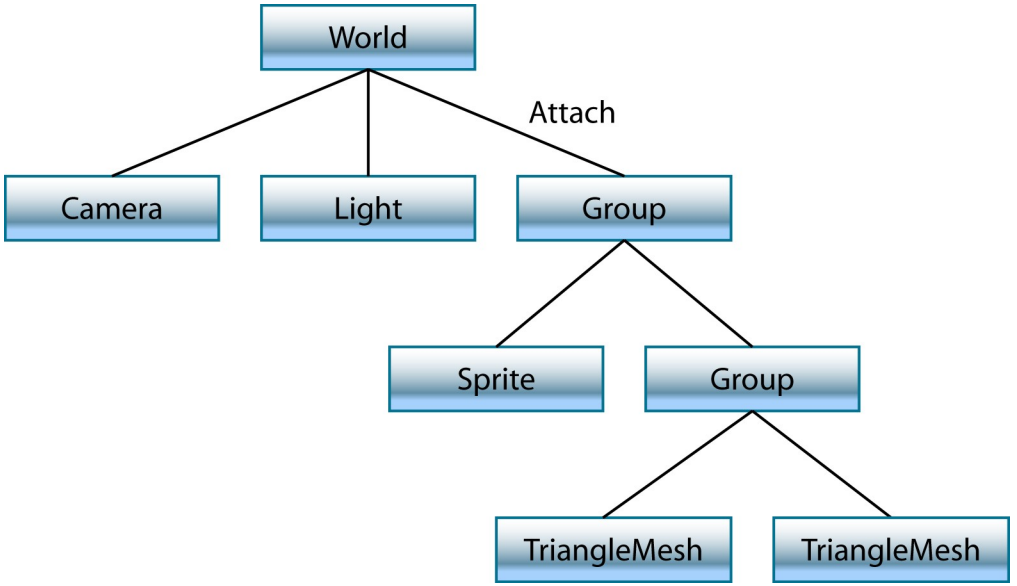


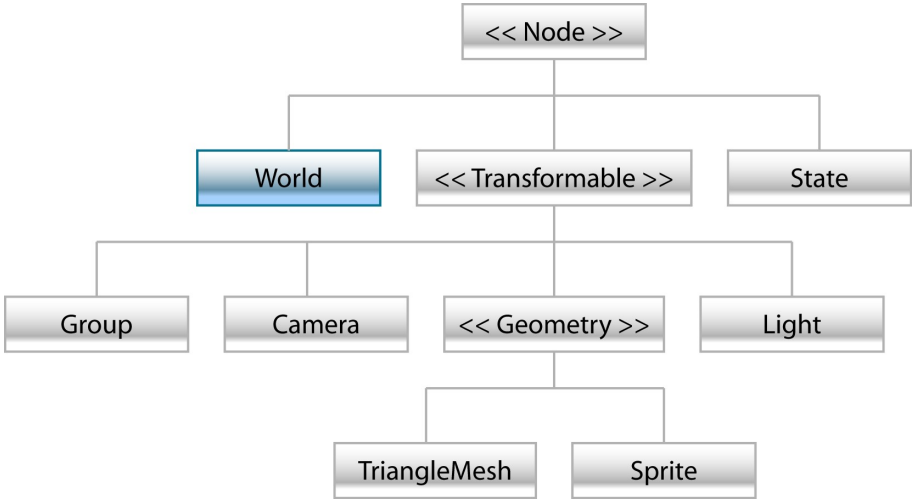
Scene graph building example

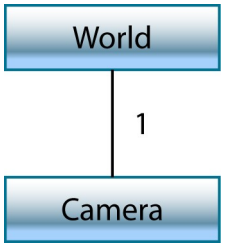




Scene graph building example







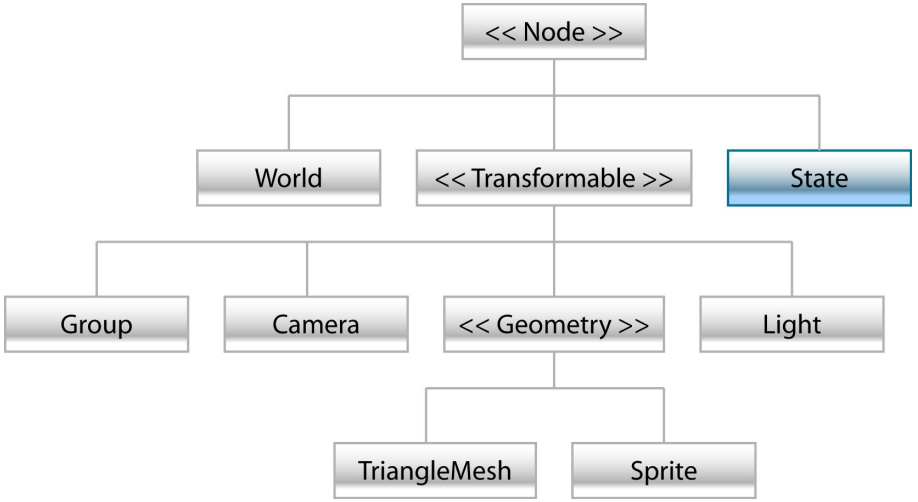
- The root node of the scene graph must be a World node. Not permitted anywhere else in the scene graph!

- Draw the entire scene graph using:

```
void drawAll(...);
```

- Must set an active camera. Render scene from this.

```
void setActiveCamera(...);
```

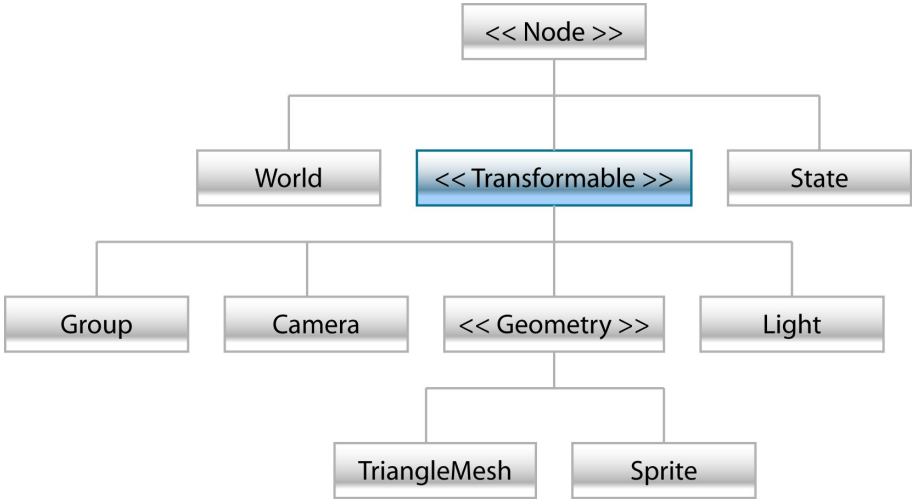


Describes a render state.

- Depth testing
- Stencil testing
- Scissors testing
- Blending
- Face culling

Don't worry about these for the assignment

A “standard state” resides in the World node. Applied before drawing begins.



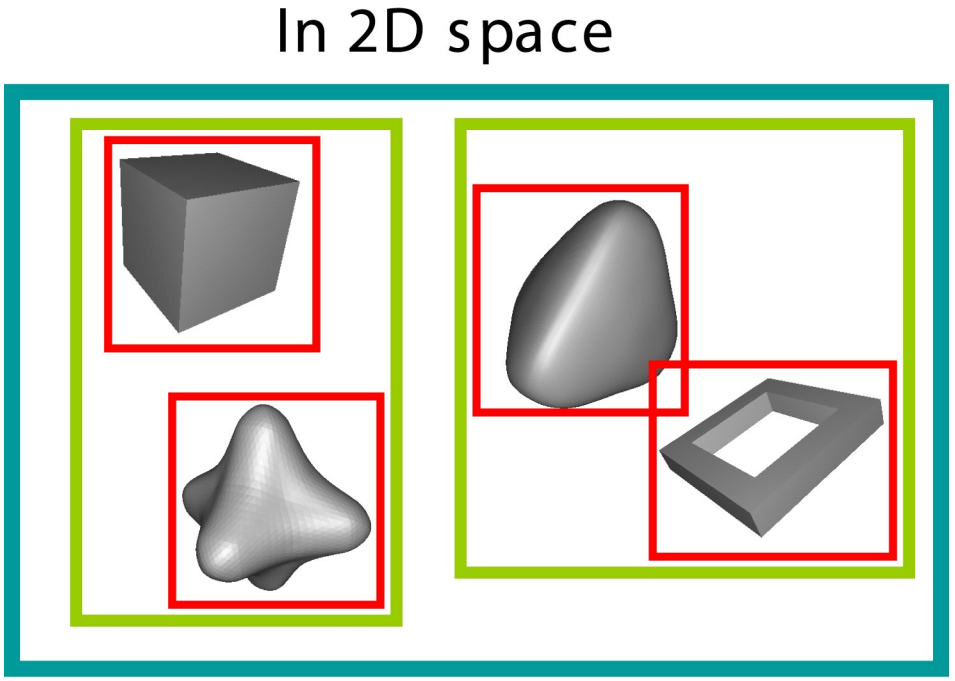
Rotate (R), Scale (S) and Translate (T)... Hierarchically!

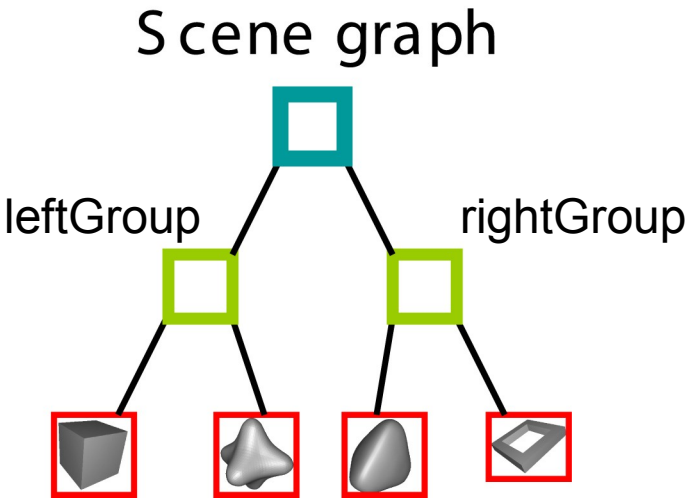
Computed as:

$$M = T * R * S$$

(think: scale, then rotate, then translate)

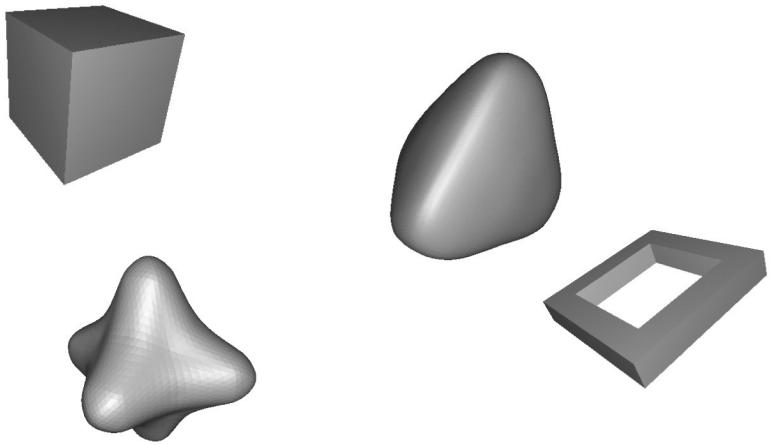
Hierarchical transformation

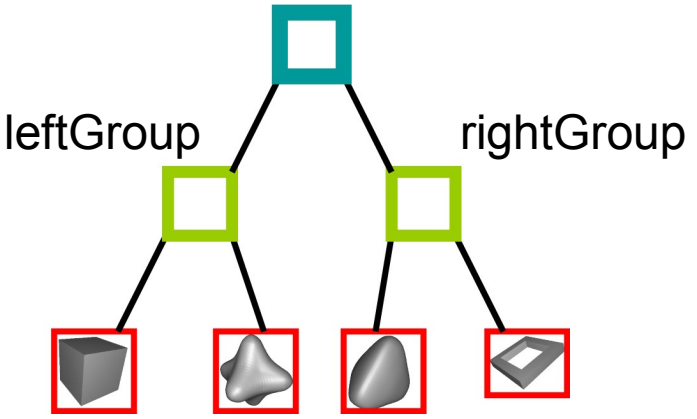




Hierarchical transformation

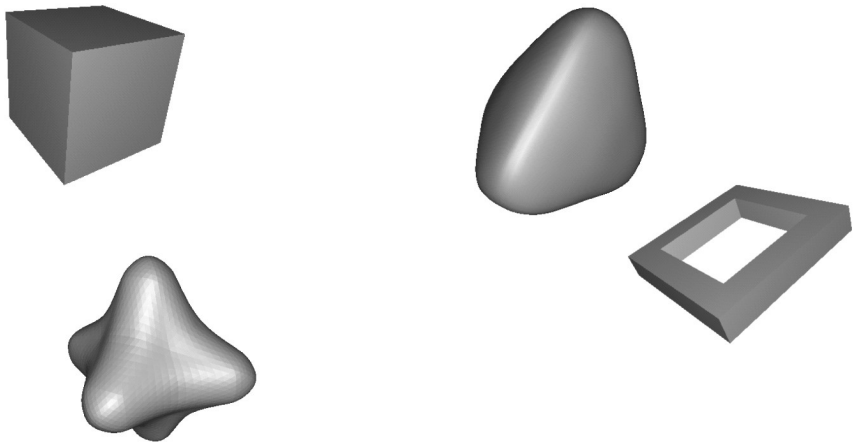
```
rightGroup->translate(2.0f, 1.0f, 0.0f);
```

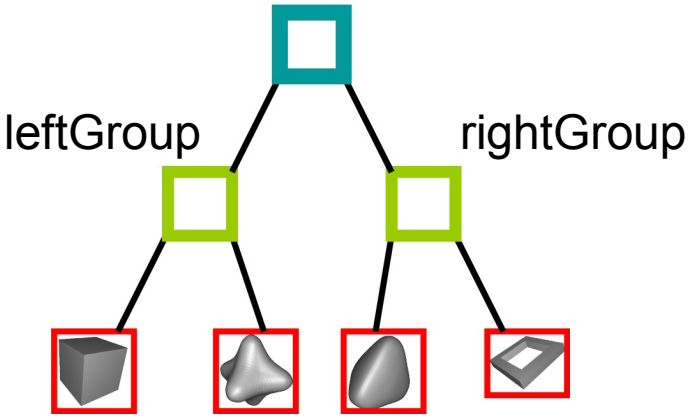


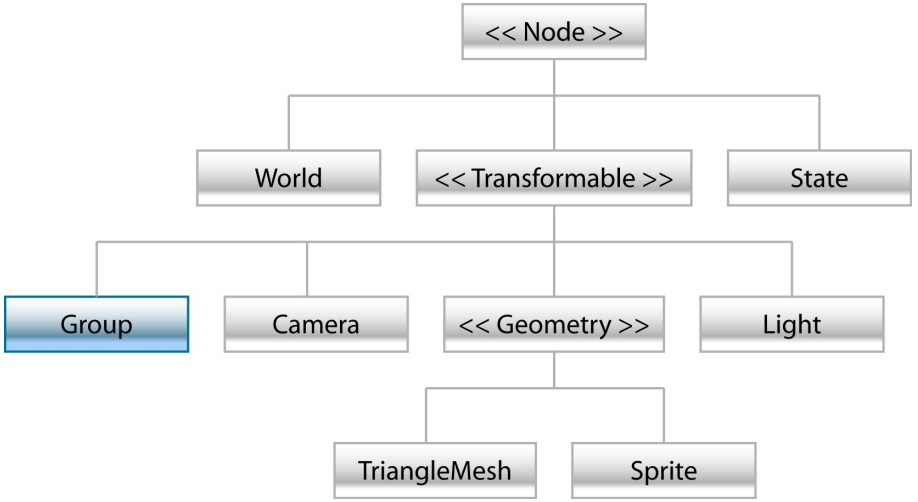


Hierarchical transformation

```
rightGroup->translate(2.0f, 1.0f, 0.0f);
```

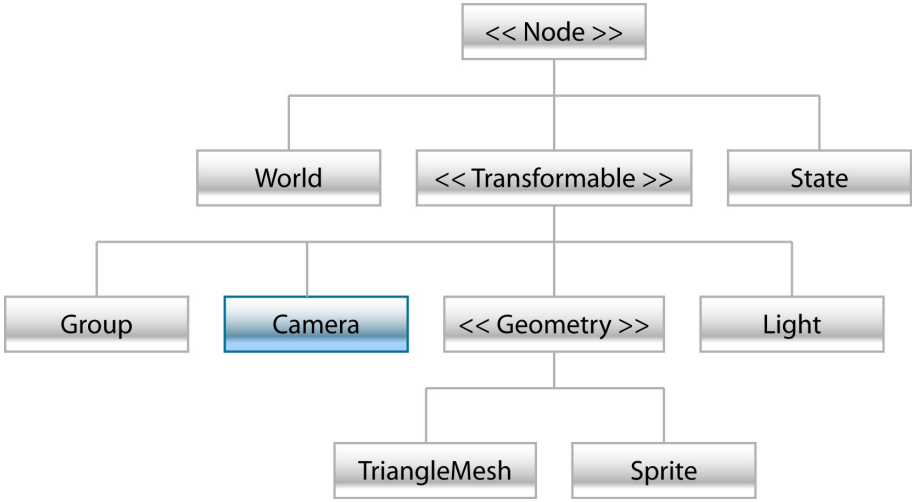






- Just a “dummy node”...
- Use to stack hierarchical transformations.
- .pwn files are loaded into Group nodes.

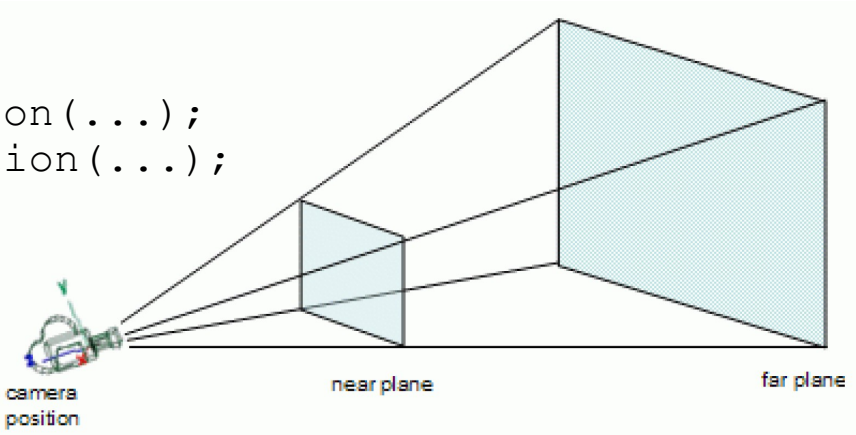
```
Group *g = sceneGraph.createGroup(“spaceship.pwn”, ...);
```

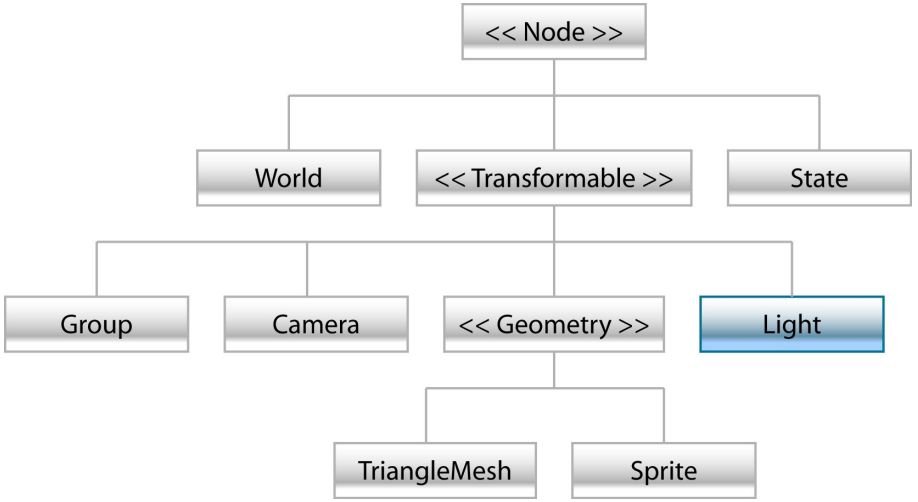


Looks down negative Z-axis

Projection is set to orthogonal, perspective or a custom matrix.

```
void setOrthogonalProjection(...);
void setPerspectiveProjection(...);
void setProjection(...);
```



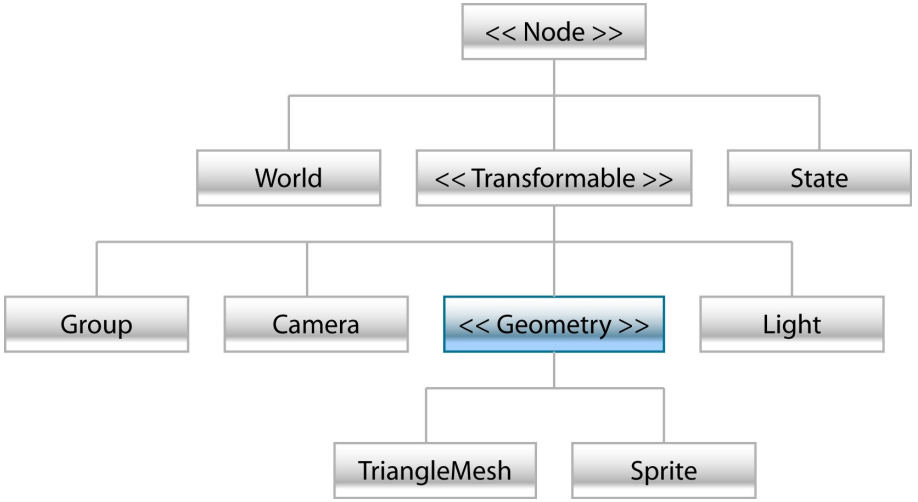


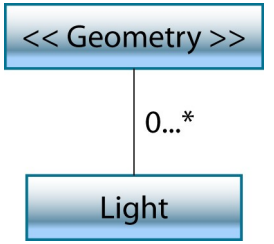
Very simple point light source.

- Color
- Intensity

A point in your scene.

Used by Geometry





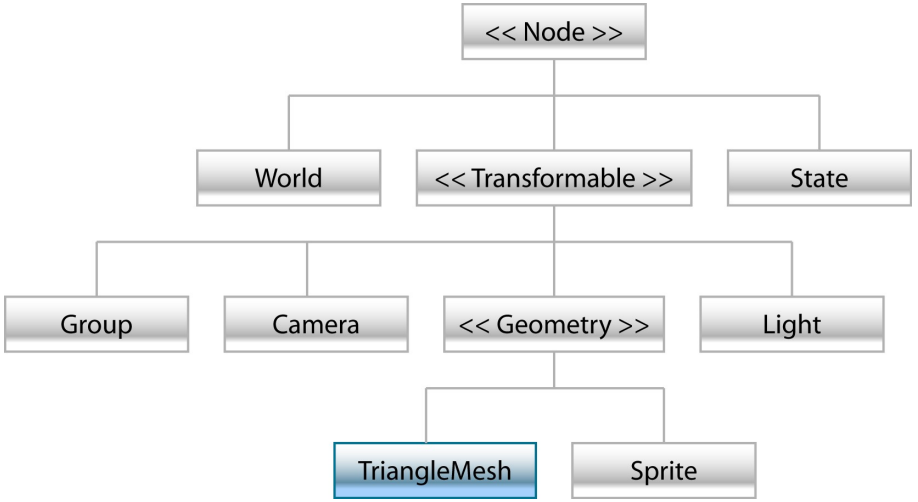
Base class of geometric objects.

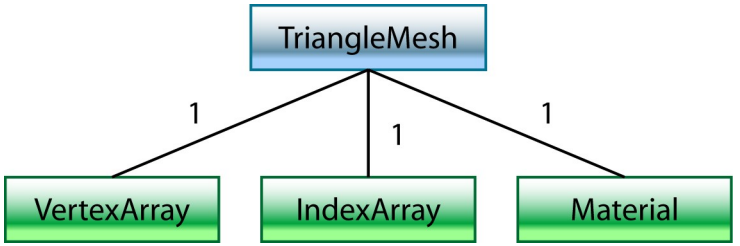
Set active lights on it for lighting calculations.

```
void setLight(...);
void clearLight(...);
Light *getLight(...);
```

It's up to the sub-classes how they use these lights.

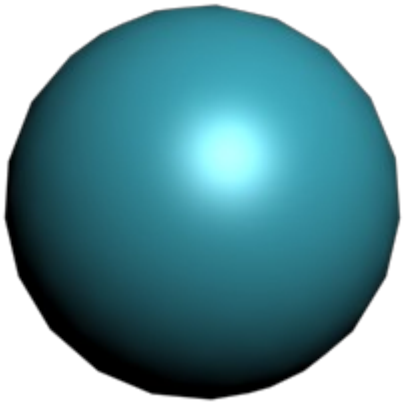
- A TriangleMesh with MaterialPhong needs at least one light.
- A TriangleMesh with MaterialColorful doesn't need any lights.

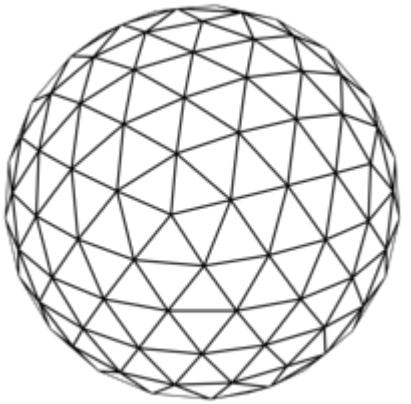


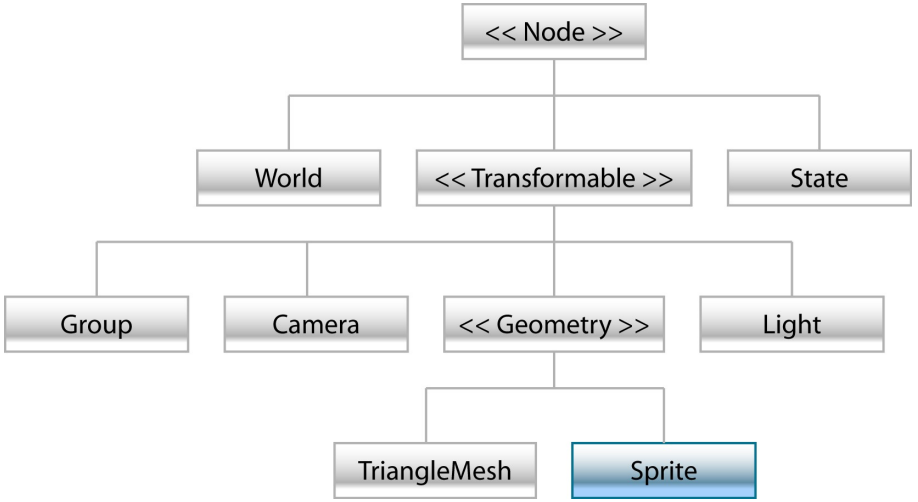


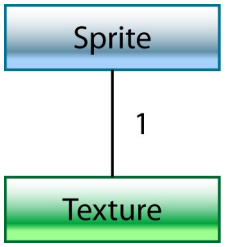
Describes a triangle mesh.

- VertexArray + IndexArray defines triangles.
- Material defines appearance.



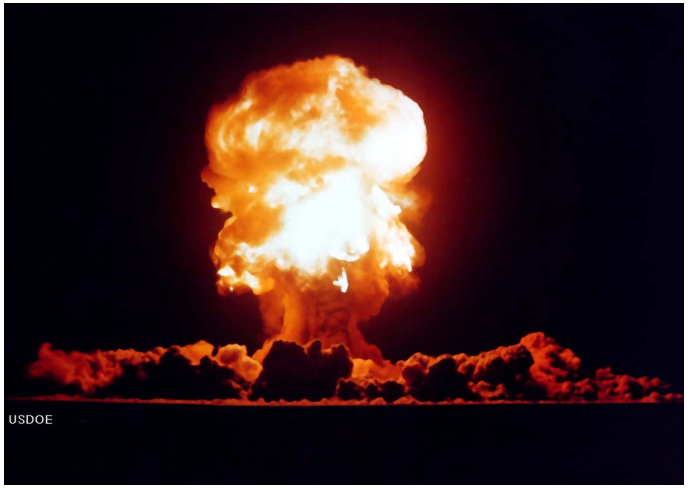






A Sprite is just two triangles glued together.

- Always faces the camera.
- *May* be lit using Light nodes.
  - No light nodes attached = fully lit
- Great for explosions!



Memory debugging

Yes it's true – no garbage collection!

Memory.h *tries* to catch mistakes...

- Writing outside of an array
- Writing to free'd memory
- Freeing already free'd memory
- Memory leaks

Enable #define MEMORY_DEBUG
from time to time...

Memory debugging

In the emulator...