Advanced programming classes #3

— IMAC 3rd year —

Animation

Part 3/5 of « Boids Tower Defense »

Objectives:

- To use existing code and documentation
- To manipulate smart pointers, callbacks and visitors
- To get into a few aspects of a scene graph library (here OpenSceneGraph)

Work to be done in two-persons teams, to be then sent to your teacher (neilb@free.fr). Exercises are independent and can be done in any order.

Exercise 1 – Animating the clouds

The application can display basic clouds, using sprites. These don't move for now; you'll have to add movement. Those clouds are ndes of the OSG scene graph, created in `makeCloud()`.

To know: When doing update traversal of the scene graph, nodes (here `MatrixTransform` and `Geode`) are visited by an `UpdateVisitor`.

1. Animate clouds with a simple horizontal movement, <u>synchronized with simulation time</u>. You will implement using an "update callback", derived from `NodeCallback` (see `osg::Node::setUpdateCallback()`). Warning: read carefully about responsibilities of a `NodeCallback`…

2. Explain when the '*update callback* is deallocated, and give the line number of `App.cpp` where it happens.

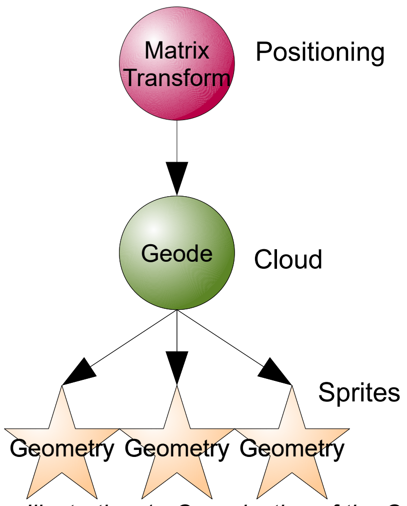

Illustration 1 : Organization of the OSG classes for a cloud

Secondary objectives

- [Easy] Add a "teleportation" of clouds when outside the map (wrap their movement so that moving too far away on the left teleport them on the right, and so on). (0,5 point)
- Explain why the animation update rate can vary (updates per second), whereas it would not if placed in `Game::step()` (as in exercise 3). (0,5 point)
- [OpenGL] Give pros and cons of *'point sprites'* compared to the sprites used here. (0,5 point)
- [Difficult] Explain briefly how you would make the clouds progressively translucent as the camera approaches them. Tell in which OSG traversal phase this would happen, as well as the visitor that should be used. (1 point)

Personal training

You may train manipulate OpenSceneGraph, for instance by...

- Animating the color of the clouds, depending on the day/night cycle (see exercise 3). You'll have to understand what clouds are made of, and thus may have to read the code creating them.
- make the clouds progressively translucent as the camera approaches them.
- [Difficult] Handle progressive formation and disappearing of clouds. You may want to create a new OSG node class, if you wish.

Exercise 2 – User traversal

To update specific nodes, you may make them "react" to an *Event*, by adding an `EventCallback` on those nodes. However, that may imply creating a lot of callbacks. And this may become difficult if nodes are not previously known.

You will thus update nodes using a custom *visitor* (see slides and web resources about visitors). Create a `NodeVisitor` which modifies all `MatrixTransform` that have the name "Cloud", by adding a translation on the Z axis. To ensure your code works:

- Name "Cloud" the `MatrixTransform` of the clouds (in `makeCloud()`, using `setName()`).
- Create clouds with an offset that push them out of view (for instance, +10 000 on Z axis).
- Apply your visitor on the scene's root, so that it neutralize the offset. You can do that in `App::run()` in order to test, just after `load()`. Clouds must then be at their usual position in scene.

Take some precautions:

- Base class (`osg::NodeVisitor`) uses `TRAVERSE_NONE` by default, and this is **not** what we want here. As you'll apply your visitor on the root node, you logically need to set the traversal mode to `TRAVERSE_ALL_CHILDREN`.
- Be aware that, depending on your choices, your "cloud teleportation" (exercise 1) may counteract what your visitor or initial positioning does. You may deactivate teleportation, or ensure it only operates on XY plane and your visitor on Z, for instance.
- You may write on the standard ouput when a cloud is visited. There are 5 clouds by default; you should have 5 messages.
- Read <u>very carefully</u> visitor's documentation…

Exercise 3 – Day/night cycle

You will implement a simple day/night cycle. The "solar" light source to simulate:

- Is a directional light source, not a point light.
- Simulates the Sun rotating around Y axis (or better: on an axis close to Y, slightly tilted on the X axis).
- Must have its color (especially diffuse) subdued when close to the horizon, **and** turned off at night.

Your implementation:

- Is to be done in `Game::updateLights()`, which is called in `Game::step()` (physical simulation iteration). `gameLightSources[LIGHT_DAY]` is already allocated; you just have to animate.
- Will estimate a global <u>perceived</u> light intensity in `Game::computeDayLightIntensity()`. This method is only there to provide an approximation to adjust special effects (explosions) lighting. This should thus not be used as a data source for `updateLights()`.

Secondary objectives

Following objectives are only bonuses. They are all optional and can be implemented in any order.

- Add a "global ambient" during night (1 point). Pay attention to the different definitions:
  - the "ambient" color component of a light source,
  - the "ambient" color component of a material, and
  - and the "global ambient" illumination, which is a constant color added everywhere.
- Make the light color become smoothly reddish at sunrise and sunset. (0,5 point)
- Add a light source to simulate the Moon. (0,5 point)
- Animate the sky texture. Sky box is actually multi-textured (one for day, one for night). You must act on the interpolation value to make a transition. Read carefully the doc, and read sky creation code (In `App::load()`). Add the update inside `Game::step()` or `Game::updateLights()`. (0,5 point)