SGCT – Simple Graphics Cluster Toolkit

SGCT is a static C++ library for developing OpenGL applications that are synchronized across a cluster of image generating computers (IGs). SGCT is designed to be as simple as possible for the developer and is well suited for rapid prototyping of immersive virtual reality (VR) applications. SGCT supports a variety of stereoscopic formats such as active quad buffered stereoscopy, passive side-by-side stereoscopy, passive over-and-under stereoscopy, checkerboard/DLP/pixel interlaced stereoscopy and anaglyphic stereoscopy. SGCT applications are scalable and use a XML configuration file where all cluster nodes (IGs) and their properties are specified. Therefore there is no need of recompiling an application for different VR setups. SGCT does also support running cluster applications on a single computer for testing purposes. SGCT is based on GLFW and GLEW and enables the user to utilize functionality from these libraries as well.

Features

- Frame synchronization, swap buffer synchronization (hardware) and application data synchronization.
- PNG texture management
- GLSL shader management
- Freetype font management
- Error messaging across the cluster
- Statistics and performance graphs
- External TCP control interface (telnet, GUI applications, Android apps, iOS apps)
- Easy thread handling (GLFW)
- Joystick and gamepad support (GLFW)
- Anti-aliasing (GLFW)

Limitations

Windows only at the moment (Visual Studio and MinGW) but will support more platforms in the future. Currently there is no tracking support in the library but VRPN support will be implemented in a later version.

Currently not all parts of SGCT supports OpenGL 4 but will be ported in a later stage of the project.

Classes

In the SGCT namespace you will find the following classes:

- Engine
- FontManager
- MessageHandler
- ShaderManager
- ShaderProgram
- SharedData
- TextureManager

Engine

This class is the core of SGCT. There is a deeper description later in this document in the “How it works” section. Window handling, network handling and rendering is done through this class.

FontManager

This is a singleton that manages, reads and renders Freetype fonts.

MessageHandler

This is a singleton that handles messages from the cluster. Messages are fetched from the slaves and printed in the master’s console for easier debugging.

ShaderManager

This is a singleton that manages, reads and sets GLSL shader programs.

ShaderProgram

With this class a user can bypass the ShaderManager and access shader programs directly.

SharedData

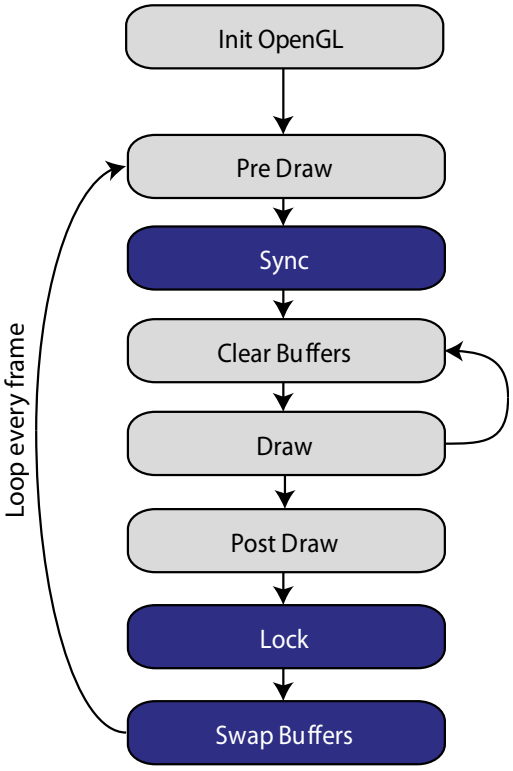
This class encodes and serializes different types (floats, doubles, integers, booleans, etc…) on the master and sends a single block of data to the slaves. The slaves use this class to decode and de-serialize the data back to its original form. The data container is dynamic and can be changed from frame to frame.

TextureManager

This is a singleton that managers, reads and sets PNG textures. It supports grayscale, alpha, RGB and RGBA textures. GPU compression, anisotropic filtering and mipmaps can be set as well.

How it works

The most important component in SGCT is the engine. The engine handles all the initiation, rendering, network communication and configuration handling. The user can bind functions (callbacks) to the engine to customize specific tasks. Callbacks for keyboard and mouse input are handled by GLFW. Bonded functions will be called in different stages in the rendering process illustrated below:



Init OpenGL

This stage is called after the OpenGL context has been created and is only called once. This callback must be set before the Engine is initiated to have any effect. During the other stages the callbacks can be set end re-set anytime.

Pre Draw

This stage is called before the data is synchronized across the cluster. Set the shared variables here on the master and the slaves will receive them during the sync stage.

Sync

This stage distributes the shared data from the master to the slaves. The slaves wait for the data to be received before the rendering takes place. The slaves also send messages to the server if there are any (console printouts, warnings and errors). There are two callbacks that can be set here, one for the master and one for the slaves. The master encodes and serializes the data and the slaves decode and de-serialize the data.

Clear Buffers

This stage clears the buffers and sets the clear color. If this callback is set then it overrides the default clear buffers function. This stage can be called twice every frame if stereoscopic rendering is active.

Draw
This stage draws the scene to the current back buffer (left, right or both). This stage can be called several times per frame if multiple viewports and/or if stereoscopic rendering is active.

Post Draw
This stage is called after the rendering is finalized.

Lock
During this stage the master is locked and waiting for all slaves. No callbacks can be set during this stage. The master doesn't lock and wait if Nvidia's swap barrier is active.

Swap Buffers
The front and back buffers are swapped. Triple buffering should not be used on a cluster when the application waits for vertical sync. No callbacks can be set during this stage. If Nvidia's swap barrier is active then the buffer swap will be synchronized using hardware across the cluster.

Bind functions
The following functions can be found in the Engine class to bind functions as callbacks:

```
void setInitOGLFunction( void(*fnPtr)(void) );
void setPreDrawFunction( void(*fnPtr)(void));
void setClearBufferFunction( void(*fnPtr)(void) );
void setDrawFunction( void(*fnPtr)(void) );
void setPostDrawFunction( void(*fnPtr)(void) );
```

There is one additional function that binds a function as callback when an external TCP control interface is being used. This callback is called when TCP data is received.

```
void setExternalControlCallback( void(*fnPtr)(const char *, int, int) );
```

The arguments above are: buffer, size of buffer and client index. The following functions can be found in the SharedData class to bind functions as callbacks during the Sync stage.

```
void setEncodeFunction( void(*fnPtr)(void) );
void setDecodeFunction( void(*fnPtr)(void) );
```

Getting started

XML configuration file

In order to run any application a valid XML configuration file that describes the VR setup must be provided. The simplest form of a cluster consists of a single node. Here is an example on how such configuration can look like.

```
<?xml version="1.0" ?>
<Cluster masterAddress="127.0.0.1">
     <Node ip="127.0.0.1" port="20401">
          <Window fullscreen="false">
               <Stereo type="none" />
               <Pos x="200" y="300" />
               <!-- 16:9 aspect ratio -->
               <Size x="640" y="360" />
          </Window>
          <Viewport>
               <Pos x="0.0" y="0.0" />
               <Size x="1.0" y="1.0" />
               <Viewplane>
                    <!-- Lower left -->
                    <Pos x="-1.778" y="-1.0" z="0.0" />
                    <!-- Upper left -->
                    <Pos x="-1.778" y="1.0" z="0.0" />
                    <!-- Upper right -->
                    <Pos x="1.778" y="1.0" z="0.0" />
               </Viewplane>
          </Viewport>
     </Node>
     <User eyeSeparation="0.069">
          <Pos x="0.0" y="0.0" z="4.0" />
     </User>
</Cluster>
```

In most 3D graphics applications a camera is defined by a vertical field of view (VFOV) and an aspect ratio. In VR you usually set up a view plane per viewport which corresponds to the physical dimensions of a screen (or channel if the screen is multi-channel). By doing so enables the use of unsymmetrical frustums which are needed for correct stereoscopy. The coordinates of the view plane and the user position are in meters and this result in that the OpenGL coordinates are in meters too. Note that the coordinate system origin can be shifted by offsetting the coordinates for the view plane and the user. In this case the origin is set to the middle of the screen.

SGCT supports only one window per node but can contain several viewports. The viewport coordinates are in normalized screen space [0.0, 1.0]. Each computer will compare its ip addresses with all nodes and the master address to determine which node configuration to use and if it's the master or not. When running several nodes on a single computer this process can be overridden by using command line arguments.

Command line arguments

Below is the list of command arguments that the Engine uses. If the final application also will use command line arguments then make sure that there are no conflicts.

- -config <path_to_configuration_file>
- -local <index to node to use settings from>
- --slave

Note that the last two commands are only used when simulating a cluster on a single computer.

Example 1, run in cluster or standalone mode:

```
application.exe -config “cluster.xml”
```

Example 2, run two nodes locally at the same computer (cluster simulation mode). The first command will start the master and the second command will start the slave:

```
application.exe -config “cluster.xml” -local 0
application.exe -config “cluster.xml” -local 1 –slave
```

First application

Creating applications using SGCT is very simple and requires minimum coding. Here is an example which just opens up a window and initiates the synchronization across the cluster.

```
#include "sgct.h"

sgct::Engine * gEngine;

int main( int argc, char* argv[] )
{
     gEngine = new sgct::Engine( argc, argv );

     if( !gEngine->init() )
     {
          delete gEngine;
          return EXIT_FAILURE;
     }

     // Main loop
     gEngine->render();

     // Clean up
     delete gEngine;

     // Exit program
     exit( EXIT_SUCCESS );
}
```

Let's draw a triangle. Create a draw function that doesn't take any arguments and doesn't return anything. Don't forget to declare the function.

```
void myDrawFun()
{
	//render a single triangle
	glBegin(GL_TRIANGLES);
		glColor3f(1.0f, 0.0f, 0.0f); //Red
		glVertex3f(-0.5f, -0.5f, 0.0f);

		glColor3f(0.0f, 1.0f, 0.0f); //Green
		glVertex3f(0.0f, 0.5f, 0.0f);

		glColor3f(0.0f, 0.0f, 1.0f); //Blue
		glVertex3f(0.5f, -0.5f, 0.0f);
	glEnd();
}
```

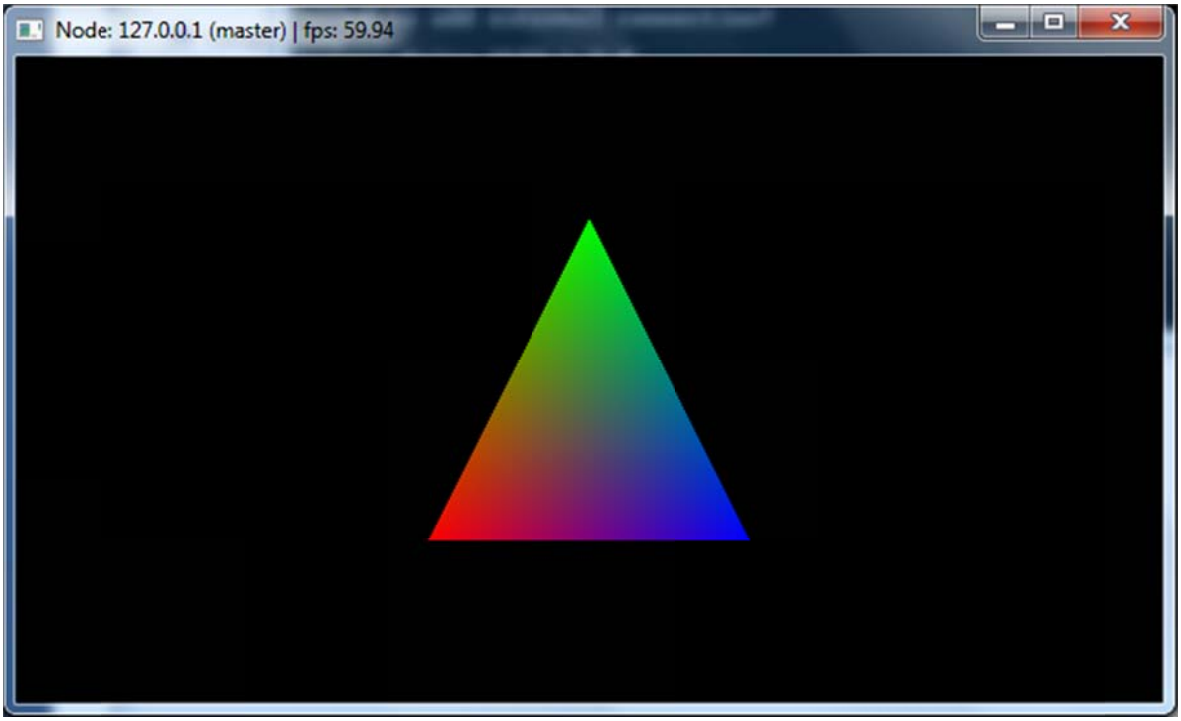
Now we need to tell SGCT to use the render function.

```
//Bind your draw function to the render loop
gEngine->setDrawFunction( myDrawFun );
```

This call can be placed anywhere between engine object creation and start of render loop. Compile and start the application with the following arguments:

```
-config "path_to_the_configfile.xml"
```

It's possible to set the arguments in the project's options in your IDE (Integrated Development Environment). Now the result should look something like the image below.



Let's make the triangle spin, synchronized across a cluster. In order to do that a variable needs to be shared between the nodes. We can use the time from the Master node and send it to all the slave nodes. The shared variables are synchronized before the draw function is called in the pre draw function. Variables sent across the network needs to be serialized, the master encodes the variables and the slaves decodes them. Since it's a serial process the order of the variables needs to be the same in the decoder as in the encoder. The code below show how to synchronize the time variable:

```
void myPreDrawFun();
void myEncodeFun();
void myDecodeFun();

double time = 0.0;

void myEncodeFun()
{
     sgct::SharedData::Instance()->writeDouble( time );
}

void myDecodeFun()
{
     time = sgct::SharedData::Instance()->readDouble();
}

void myPreDrawFun()
{
     if( gEngine->isMaster() )
     {
          time = glfwGetTime();
     }
}
```

Set/bind the functions to SGCT by:

```
gEngine->setPreDrawFunction( myPreDrawFun );
sgct::SharedData::Instance()->setEncodeFunction(myEncodeFun);
sgct::SharedData::Instance()->setDecodeFunction(myDecodeFun);
```

Use the time variable to rotate the triangle around the Y-axis in your draw function before you draw the triangle:

```
float speed = 50.0f;
glRotatef(static_cast<float>( time ) * speed, 0.0f, 1.0f, 0.0f);
```

Download the source code and project files at the c-student wiki.