Basic Camera

Today we are going to create our own class to handle basic camera functionality.

Task 1 – Vector3D

You need to ensure to implement operator overloads for the Vector3D struct. You will require the following operator overloads:

```
	Vector3D operator* (float scaler)
	{

		return Vector3D(x * scaler, y * scaler, z * scaler);
	}

	Vector3D operator+ (const Vector3D& other)
	{
		return Vector3D(x + other.x, y + other.y, z + other.z);
	}

	Vector3D operator+= (const Vector3D& other)
	{
		x += other.x;
		y += other.y;
		z += other.z;
		return *this;
	}

	Vector3D operator-= (const Vector3D& other)
	{
		x -= other.x;
		y -= other.y;
		z -= other.z;
		return *this;
	}
```

Task 2 – Camera

Add a “camera.h” header file to your solution and add the following code:

```
#ifndef _CAMERA_H
#define _CAMERA_H

#include "Commons.h"
#include <SDL.h>

class Camera
{
	//---------------------------------------------------------------------------
---------------------------
public:
	Camera();
	~Camera();
	static		Camera* GetInstance();
	void		Update(float deltaTime, SDL_Event e);
	void		Render();

	//---------------------------------------------------------------------------
---------------------------
```

```
private:
	Vector3D position = Vector3D(0, 0, 10);
	Vector3D forward = Vector3D();
	Vector3D up = Vector3D();
	Vector3D right = Vector3D();

	// horizontal angle : toward -Z
	float yaw = 3.14f;
	// vertical angle : 0, look at the horizon
	float pitch = 0.0f;
};

#endif //_CAMERA_H
```

Add a “camera.cpp” source code file to your solution and add the following code:

```
#include "Camera.h"
#include "Constants.h"
#include <math.h>
#include "../gl/glut.h"

static Camera* instance = 0;
static float moveSpeed = 1.0f;
static float lookSpeed = 0.1f;

Camera::Camera()
{
}

Camera::~Camera()
{
}

Camera* Camera::GetInstance()
{
	if (instance == 0)
	{
		instance = new Camera();
	}

	return instance;
}


void Camera::Update(float deltaTime, SDL_Event e)
{
	// Forward Vector: Spherical coordinates to Cartesian coordinates
conversion (also known as the ‘look’ direction)
	forward = Vector3D(
		cos(pitch) * sin(yaw),
		sin(pitch),
		cos(pitch) * cos(yaw));

	// Right vector
	right = Vector3D(
		sin(yaw - 3.14f / 2.0f),
		0,
		cos(yaw - 3.14f / 2.0f));

	// Up vector : perpendicular to both forward and right, calculate using
the cross product
	up = Vector3D((right.y*forward.z) - (right.z*forward.y),
```

```
                (right.z*forward.x) - (right.x*forward.z),
                (right.x*forward.y) - (right.y*forward.x));

        //Event Handler.
        if (e.type == SDL_KEYDOWN)
        {
                switch (e.key.keysym.sym)
                {
                case SDLK_w:
                        //move forwards
                        position += forward * moveSpeed;
                        break;

                case SDLK_s:
                        //add move backwards code using the forward vector
                        break;

                case SDLK_d:
                        //add strafe right code using the right vector
                        break;

                case SDLK_a:
                        //add strafe left code using the right vector
                        break;

                case SDLK_UP:
                        //look up
                        pitch += lookSpeed;
                        break;

                case SDLK_DOWN:
                        //add look down code by adjusting the pitch
                        break;

                case SDLK_LEFT:
                        //add look left code by adjusting the yaw
                        break;

                case SDLK_RIGHT:
                        //add look right code by adjusting the yaw
                        break;
                }
        }
}

void Camera::Render()
{
        Vector3D lookatPos = position + forward; // make sure we're always
looking at a point infront of the camera
        glLoadIdentity();
        gluLookAt(position.x, position.y, position.z, lookatPos.x, lookatPos.y,
lookatPos.z, up.x, up.y, up.z);
}
```

You will need to implement the code required for the missing strafe and look directions that are adjusted in the key event handler.

You will also need to add calls to get the camera instance and call the update and render function from the game screen level