Grafika komputerowa

INSTRUKCJA DO LABORATORIUM 4: obiektowa reprezentacja danych

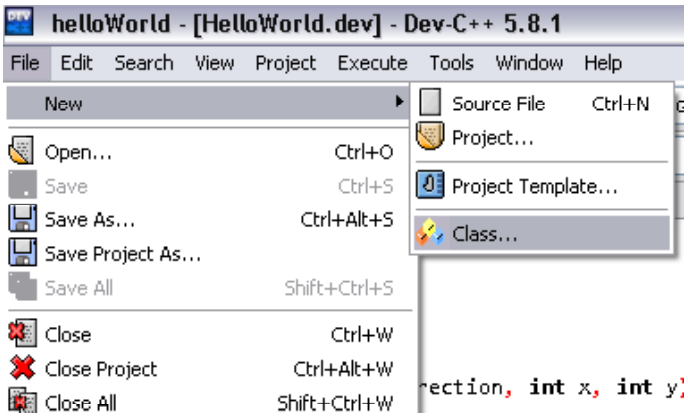
mgr inż. Paweł Stawarz
Rzeszów, 2015

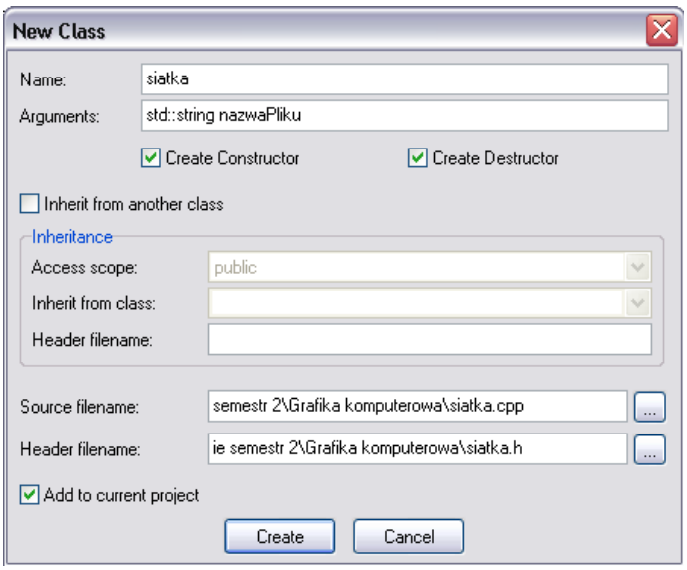
CEL ĆWICZENIA

Celem ćwiczenia jest analiza proponowanych rozwiązań z zakresu programowania obiektowego. Rozwiązania te powinny być zmodyfikowane na własne potrzeby i włączone do własnego projektu.

KLASY I OBIEKTY

Język C++ jest językiem wieloparadygmatowym (szczególnie od momentu wprowadzenia w standardzie c++0x wielu funkcji służących do metaprogramowania), jednak przeważnie stosuje się go przy programowaniu obiektowym Większość środowisk ułatwia więc użytkownikom tworzenie i pracę z klasami.





OBIEKT W PRZESTRZENI

W poprzedniej instrukcji omówiona została przykładowa definicja klasy modelu. Gdy jednak zastanowimy się – nie wszystko, co występuje w naszych aplikacjach jest modelem. Istnieją pewne cechy wspólne pomiędzy modelami, kamerą, oświetleniem i innymi obiektami. Cechy te sprawiają, że warto zaprojektować bazową klasę abstrakcyjną.

Plik nagłówkowy takiej klasy może wyglądać następująco:

```
#ifndef ENTITY_H
#define ENTITY_H

class entity
{
	public:
		void position(		const float newX,
					const float newY,
					const float newZ);
		float getXPosition() const;
		float getYPosition() const;
		float getZPosition() const;

		void rotation(		const float newX,
					const float newY,
					const float newZ);
		float getXRotation() const;
		float getYRotation() const;
		float getZRotation() const;

		virtual void display() = 0;

		entity();			// Konstruktor
		~entity();			// Destruktor

	protected:
		float pX, pY, pZ;		// pozycja w przestrzeni 3D
		float rX, rY, rZ;		// obrot w przestrzeni 3D
};

#endif
```

Definicje funkcji (które w tym wypadku powinny być umieszczone w pliku ENTITY.CPP), nie zostawią przedstawione, gdyż są trywialne. Funkcja DISPLAY jest funkcją ***czysto wirtualną*** i nie należy jej definiować (nie miałoby to zresztą większego sensu).

SCENA

W grafice 3D często wprowadza się pojęcie sceny. Sceną – w uproszczeniu - nazywamy zbiór obiektów, które w danej chwili wyświetlają się w oknie programu. W najprostszym więc ujęciu scena, to nic innego jak wektor obiektów graficznych, które można wyświetlić na ekranie:

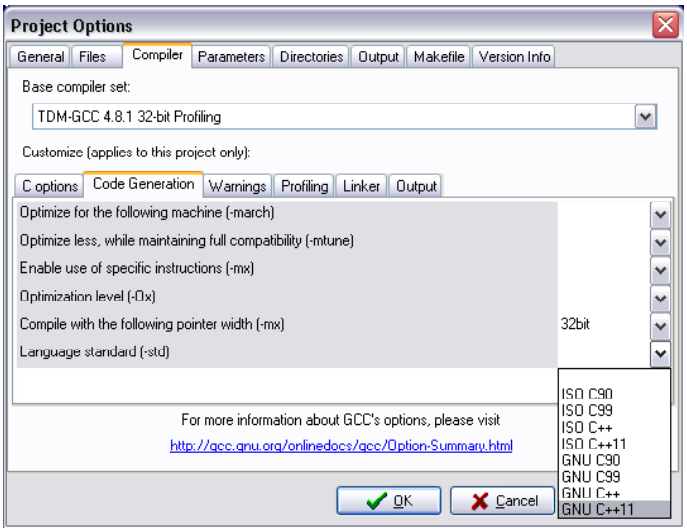
```
#ifndef SCENE_H
#define SCENE_H

#include <vector>
#include <functional>
#include "entity.h"

class scene
{
	public:
		void addObject(entity& object);
```

```
            void display();

            scene();
            ~scene();
      protected:
            // Lista obiektow
            std::vector<std::reference_wrapper<entity> > objects;
};

#endif
```

Przykładowa implementacja metod tej klasy mogłaby więc wyglądać następująco:

```
#include "scene.h"

void scene::addObject(entity& object){
      objects.push_back(object);
};

void scene::display(){
      for(entity& object: objects)
            object.display();
};

scene::scene(){}

scene::~scene(){}
```

Warto wspomnieć, że powyższy kod korzysta z funkcji dostępnych od standardu C++0x. Jeżeli występują błędy kompilacji, warto sprawdzić ustawienia kompilatora.



Zdarza się jednak, że takie przedstawienie sceny nie wystarcza. Może się okazać, że niektóre obiekty na scenie są od siebie zależne. Wtedy lepiej jest zorganizować scenę na zasadzie drzewa bądź grafu. Aby uzyskać drzewo, wystarczy do klasy ENTITY dodać wektor „dzieci" i zmodyfikować funkcję DISPLAY klas dziedziczących po tejże klasie tak, aby przy okazji wyświetlać obiekty zależne.

KAMERA

Również kamerę można oprzeć o naszą klasę bazową. W zależności od upodobania, można delikatnie zmodyfikować klasę sceny tak, aby ona przechowywała kamerę (jedną, lub wiele, jeżeli planujemy opcję przełączania się pomiędzy różnymi widokami).

Zawsze należy pamiętać, by przekształcenia związane z kamerą dokonywać ***przed*** rysowaniem obiektów na scenie.

```
#ifndef CAMERA_H
#define CAMERA_H

#include "entity.h"

class camera: public entity
{
	public:
		void display();
	protected:
};

#endif
```

Oczywiście funkcję DISPLAY należy zdefiniować w pliku CAMERA.CPP, używając funkcji służących do operacji macierzowych (w szczególności translacji i rotacji). Sposób i kolejność transformacji przestrzennych jest zależna od preferowanego sposobu sterowania kamerą. Przykładowo, jeżeli chcemy uzyskać widok izometryczny, prawdopodobnie nie będziemy obsługiwać rotacji.

MODEL

Klasa MODEL przedstawiona na poprzednich laboratoriach może zostać łatwo zmodyfikowana tak, by można było ją umieścić na naszej scenie – wystarczy zaznaczyć, że jej klasą bazową, będzie klasa ENTITY (i oczywiście zdefiniować funkcję DISPLAY). Dobrym pomysłem jest także dodanie konstruktora, który przyjmowałby nazwę pliku, z którego należy model załadować.

```
#include <vector>
class model: public entity
{
	protected:
		struct point{
			float x, y, z;
		};
		struct polygon{
			std::vector<point> vertices;
		};
		std::vector<polygon> mesh;

	public:
		model(const std::string &file);
		void display();
};
```

Takie podejście jednak średnio się sprawdzi w momencie, w którym wiele różnych obiektów na scenie dzieli tą samą reprezentację graficzną, gdyż niepotrzebnie zajmujemy pamięć RAM, wielokrotnie wczytując ten sam plik. Typowym rozwiązaniem tego problemu jest wydzielenie reprezentacji siatki do osobnej klasy i stworzenie klasy, która „zarządzałaby" alokacją pamięci.

Jeżeli byśmy postanowili skorzystać z tego rozwiązania, klasę model należałoby stworzyć klasę MESH:

```
#ifndef MESH_H
#define MESH_H

#include <string>
#include <vector>

class mesh
{
	public:
		void display();

		void load(const std::string &file);
		mesh(const std::string &file);

	protected:
		struct point{
			float x, y, z;
		};
		struct polygon{
			std::vector<point> vertices;
		};
		std::vector<polygon> structure;
};

#endif
```

Której metody realizowałyby to samo, co poprzednio metody klasy MODEL. Dodatkowo należy stworzyć klasę zajmującą się zarządzaniem siatkami:

```
#ifndef MESHLOADER_H
#define MESHLOADER_H

#include "mesh.h"
#include <map>

class meshLoader
{
	public:
		unsigned int request(const std::string &file);
		void display(const unsigned int id);

	protected:
		std::map<std::string, unsigned int> fileMap;
		std::vector<mesh> loadedMeshes;
};

#endif
```

Metoda REQUEST zwracałaby nam identyfikator siatki. Przykładowa implementacja tej metody mogłaby więc wyglądać następująco:

```
unsigned int meshLoader::request(const std::string &file){
	if(fileMap.count(file))
		return fileMap[file];
	else{
		mesh nowaSiatka(file);
		loadedMeshes.push_back(nowaSiatka);
		fileMap[file] = loadedMeshes.size()-1;
		return loadedMeshes.size()-1;
	}
}
```

Natomiast klasę MODEL należałoby zmodyfikować tak, aby nie przechowywała siatki obiektu, a jedynie identyfikator zwracany przez funkcję REQUEST:

```
#ifndef MODEL_H
#define MODEL_H

#include "entity.h"
#include "meshloader.h"

class model: public entity
{
	protected:
		static meshLoader loader;
		unsigned int meshID;

	public:
		model(const std::string &file);
		void display();
};

#endif
```

Z oczywistych względów należałoby także zmienić działanie funkcji DISPLAY oraz konstruktora. Skoro samym wyświetlaniem siatki zajmuje się klasa MESH, a ładowaniem klasa MESHLOADER, przykładowa implementacja klasy MODEL mogłaby wyglądać następująco:

```
#include "model.h"
#include <GL/freeglut.h>

meshLoader model::loader = meshLoader();

model::model(const std::string &file){
	meshID = loader.request(file);
}

void model::display(){
	glPushMatrix();
		glTranslatef(pX,pY,pZ);
		glRotatef(rX, 1.0f, 0.0f, 0.0f);
		glRotatef(rY, 0.0f, 1.0f, 0.0f);
		glRotatef(rZ, 0.0f, 0.0f, 1.0f);
```

```
            loader.display(meshID);
        glPopMatrix();
}
```

Jeżeli w naszym programie chcemy animować obiekty, klasa MODEL powinna dodatkowo przechowywać aktualną klatkę animacji danego obiektu i przesyłać ją do siatki w momencie wywołania funkcji LOADER.DISPLAY(MESHID, FRAME).

INFORMACJE DO ZAPAMIĘTANIA

Informacje te pojawiły się w trakcie wykonywania ćwiczeń i są bardzo istotne, aby móc uczestniczyć w dalszych laboratoriach i rozumieć je.

- Różnica między klasą a obiektem,
- Rożnica między deklaracją i definicją klasy,
- Pojęcie sceny, formy jej reprezentacji i związana z nią terminologia (graf sceny, obiekty na scenie),
- Idea stojąca za stworzeniem klasy zarządzającej zasobami w programach wykorzystujących grafikę trójwymiarową.