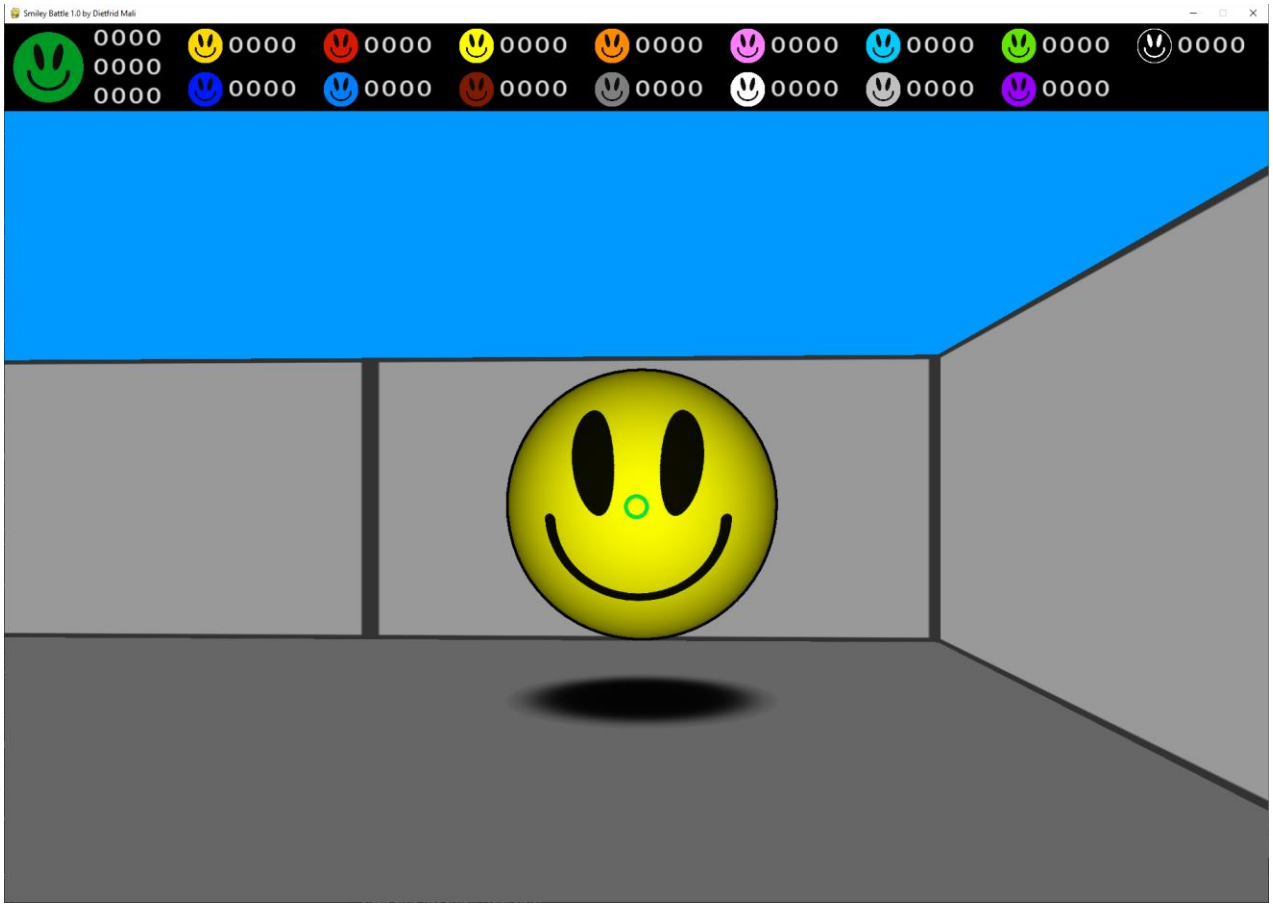
Smiley Battle

by

Dietfrid Mali



About

Smiley Battle is a remake of Midimaze, which was probably the first first-person multiplayer shooter with fluid movement. It ran on a ring network of up to 16 Atari STs which were connected via their midi out and midi in ports. Players roam a mazelike map with their smiley avatars and throw color bags at each other. Receiving three hits within a sufficiently short time puts a player in "limbo" for a brief time before he respawns at a random position. Scoring a hit yields one point, sending a player to limbo yields another point. Players start with three hit points. When hit, hit points regenerate after a short time unless the player is hit again.

Motivation

I wrote Smiley Battle as a small Python exercise to explore the language and learn its strengths and weaknesses. It uses very limited OpenGL features (the most advanced are VAOs, VBOs and three very simple shaders to render cubemaps on spheres). It should run on integrated graphics and does so on my five year old notebook.

Limitations

This implementation of Smiley Battle is purely multiplayer. There is no singleplayer mode. If you want to shoot other smileys outside of a multiplayer session with human-controlled opponents, you can set a number between 1 and 15 for the "dummies" entry in smileybattle.ini. This will cause practice smileys to appear on the map. They will just sit around and do nothing. They are mainly meant for testing purposes – to see, whether collision and hit detection and scoring work properly. Writing a decent bot AI for Smiley Battle would have gone past the scope of this project, and a bad bot AI would just mean spending time on something that would not constitute a significant improvement over sitting ducks.

Installation

Smiley battle requires numpy, pygame, pyopengl and pyopengl-accelerate. To build pyopengl-accelerate, you will unfortunately have to install MS Visual Studio and its C++ application building components. That's however all you need to do; It just needs to be present on your computer to install pyopengl-accelerate (which will be built from C source during installation).

Game Settings

Smiley battle does not offer an interactive UI right now. All game settings can be controlled via command line or a file named "smileybattle.ini" that has to be placed in the folder where smileybattle.py resides. See the enclosed file for a reference.

Input Devices

Smiley Battle supports gamepads and keyboard. Control movement with W-A-S-D, arrow keys or numpad 4-5-6-8 keys. Fire with spacebar. Alternatively, control movement with the gamepad sticks and shoot with its fire buttons.

How To Establish A Multiplayer Game Session

One player needs to act as game host. He will coordinate participation of all other players. The game host needs to tell all other participants his external IP address and the UDP port number his game client is listening on. This port number is specified with the "inPort" setting in smileybattle.ini. All other players need to enter the game host's external IP address as the "hostAddress" setting and the game host's listen port as the "hostPort" setting.

As soon as the game host's game client is started, it will start to listen for incoming connection requests by other players and will handle them one by one by sending them the map, game settings, list of players already in the game and any other stuff they need to know. Once synchronization of a new player is done, he will appear on the map and can start to participate in the battle.

Every player needs to enter his own external IP address as the "localAddress" setting in smileybattle.ini. As long as they are not sharing the same external IP address (e.g. because they are behind a common DSL router), every player can use the same in and out ports; They just need to make sure they are not used by another application running on their system. The port range 9100 – 9200 should usually be safe to use. If two players share an external IP address, they can only be distinguished from outside by their port numbers, so they need to chose different in and out ports from each other.

Code

Smiley Battle consists of around 4.700 lines of Python code (counted with *pygount --format=summary*; You can *pip install pygount*). I have written all of the code myself. The DACS (Dijkstra Address Calculation Sort) based shortest path search is a port of a module I wrote in C++ years ago. All other code has been created from scratch. I looked a few things up on the internet, mostly how to texture spheres with cubemaps and also found some elegant icosphere creation code in C++ which I took as a blueprint for my implementation (I had however created similar code myself a couple of years ago). Building the game including creating all textures took me around five weeks.

You can find an iterator in vertexdatabuffers.py and a lambda function in renderer.py. networkhandler.py contains a docstring following best practice guidelines for Python docstring use I found on the internet. icosphere.py uses recursion. I did not use Python's getter and setter functionality since for my use cases I didn't see an added value and you don't get true encapsulation in Python anyway.

Something causing real headaches was Python's aliasing, which I had to find ways around. The worst case was aliased lists in my vertex data buffer implementation, which led to data of different types being stored in the same

list. I found a way to work around that by slicing an empty list and assigning it to the various buffer labels.

Pythons indenting syntax combined with its flexible type handling caused some problems on a few occasions, where I had accidentally indented a final return statement, leading to functions returning None instead of a meaningful value.

I did not dig into decorators, which look like quite an interesting concept. However, I didn't really find an application for them in Smiley Battle.

C++ and C# Versions

There is also a C++ port of the software I built after finishing the Python version, which took me another three weeks. The C++ version has around 4115 lines of C++ source code and 3075 lines of header code lines (not counting comments, blank lines, and single-line closing curly braces), which corresponds well with the around 7150 code statements of the C++ version.

Porting to C++ admittedly was a pain in the neck on more than one occasion, since everything Python does automatically you have to do manually in C++. On the other hand, C++ gives you a lot more control over what compiler and code do.

Porting from C++ to C# was a breeze, and getting the C# version to run after the code port was finished took less than two days. The C# version has around 8.500 code lines (not counting comments, blank lines, etc.) and around 5260 statements.

Assets

Smiley Battle uses very few textures (it actually only needs the black and white smiley textures for the players; All the other colors are not needed anymore, but are kept because you never know. 😉)

I have created all textures myself, using GIMP for the purpose.

The sounds are public domain and stem from an old 3D game I had maintained for many years (Descent 2 / D2X-XL). They aren't perfect, but they do the job. I would have liked more cartoonish and friendlier sounds, but sound design is beyond my expertise.

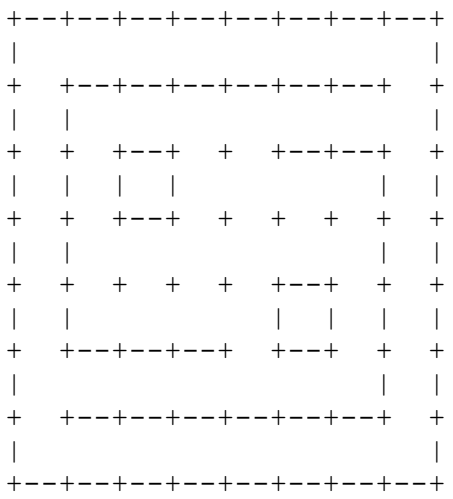
Geometric Objects

The only three-dimensional shapes used in this game are spheres, which are generated from a cube, and a torus with a rectangular cross section. Smiley Battle has proper collision handling, so you can bump into walls and push other players around.

Map Format

A map consists of corridors and walls. It is described in a simple text format. Horizontal walls consist of two consecutive minus signs ("--"), vertical walls consist of a vertical stroke ("|"), and a corridor space consists of two consecutive blank spaces (" "). Between each map element, a plus sign ("+") for better visual presentation can be inserted.

A sample map looks like this:



Unlike the original map format (which I lacked a good description for to use it), this is quite an easily readable format, and the text maps match the ground plan of an in-game map quite well.

Game Loop

The game loop

- runs while the controls handler doesn’t signal program termination by the user
- computes physics-based changes to actors (speed and direction changes, collisions)
- renders the scene
- updates sound sources
- gathers and replies to network messages and updates game and actor states depending on message contents
- cleans up deleted actors

Rendering

Smiley Battle uses a very simple OpenGL (v330 core) based renderer with only the most basic OpenGL features (the most advanced are probably VAOs, VBOs and a few shaders). Nevertheless, a GPU with a driver capable of supporting these OpenGL features is required. Even though Smiley Battle runs on my ancient notebook, for some integrated GPUs this does not seem to be the case.

Collision Handling

Smiley Battles distinguishes four types of collisions:

1. Player - player
2. Player – projectile
3. Player – wall
4. Projectile – wall

Players bounce off each other and walls (at which they may also slide along). When a player bounces off another player, he may hit a wall and bounce back. This is handled by looping the collision handling a few times. Players having bounced off a wall will be stationary in subsequent collision handling loops, i.e. will not move when hit again by a player – that player will then be moved exclusively, to resolve the collision.

Sound Handling

Pygame’s sound and channel management does not provide the means to identify individual channels (e.g. by channel id) and specifically deal with such a channel (when e.g. the corresponding sound source disappears). You also cannot just allocate a bunch of channels and use them as you need them: pygame will provide sound channels non-transparently, just by handing over any currently unused channel. Smiley Battle’s sound handler therefore provides an interface where each sound channel the game uses gets its individual id and can be tracked that way.

Pathfinding

Some "fancy" stuff I included is a DACS (Dijkstra Address Calculation Sort) based shortest path search to quickly compute distances between movement nodes the application's map handler creates. Right now, the only purpose is to approximate the distance between the viewer and sound sources for proper spatial sound volume computation. Unfortunately, even with a highly optimized algorithm like DACS this tends to be slow with Python even on a fast computer.

Networking

Smiley battle has a simple UDP based networking interface to allow multiplayer matches over the internet. The networking model is peer to peer (see networkhandler.py for a brief documentation). The clients communicate with each other with simple text-based messages (I might switch this to binary packed data using Python structs … later). Every client manages his local actors (i.e., his avatar and any shots he fires). He periodically reports their positions, heading, size, hit points etc. to every other player, who will update that player's actors on their client accordingly.

Integrating a new player is controlled by a game host. The game host is one of the players. Smiley Battle identifies the game host by the host address provided via the command line or via smileybattle.ini. A match can proceed when the game host disconnects for some reason, but if that happens, players cannot join or rejoin the match.

Once a player has successfully joined a match, he periodically broadcasts updates about his position and heading and the position and heading of all shots fired be him to the other players.

Each player needs to provide his IP address to the game via command line or smileybattle.ini. Failing to do so will result in the game not being able to open UPD sockets and hence make it impossible for it to communicate with other players or the game host via the internet. When playing in a local network (e.g. behind a router at home), you can use your local network addresses (usually something 192.168.178.XXX).

Listening for network data runs in a separate thread. This can be switched to polling with the parameter *multithreading={0|1}*.

Network Protocol

Smiley Battle has a simple UDP based network protocol. CNetworkHandler implements four groups of functions:

1. Helper functions
2. 1:1 Send functions
3. Message processing functions
4. Broadcasting functions

The send functions are mostly used for integrating a new player in a match. The control flow of this process is as follows:

Client	Host
send join request →	
	← send enter request
add host to player list	
request map data →	
	← send map data
create internal map data	
request game parameters →	
	← send game parameters
apply host's game parameters	
request player data →	
	← send data of all players
add players to actor list	
Send enter request to others player →	
	other players add new player to their actor lists
request projectile data →	
	← send data of all live projectiles
add projectiles to actor list	
send enter request →	
	assign position and heading to new player
	← send player position and heading
start respawn sequence	
send (spawn) animation state →	
	spawn new player
← send updates →	
	← send updates →
send leave →	
	remove player and his child actors from actor list
quit game	

The network protocol offers some robustness by repeating join related requests until they are properly fulfilled or the game host timed out, in which case the join process is started over.

Every player periodically sends updates of his actors' position, heading and life to each other player. However, shots fired may get lost when a player experiences packet loss, since a fire event is only sent to each other player once.

The network code also checks for player timeouts by keeping track of the time a player sent his last packet and comparing that to the current time.

Performance

With a single player, Python Smiley Battle runs at close to 200 fps on the test system (Intel Core i7 9700K, Nvidia Geforce RTX 2070) at a resolution of 1920x1080. With 16 players, the frame rate drops to around 20 fps. The C++ version reaches around 4500 fps with a single player and 1500 fps with 16 players. Since the rendering is fairly optimized, this difference is mostly due to CPU load: With many players, there are a lot of calculations on the CPU, like collision handling and sound computation. This is much more costly with Python than with C++. With around 3600 fps with a single player, the C# version is about 20% slower than the C++ version. The performance loss would certainly be much more tangible with a full blown 3D game with hundreds or thousands of in-game objects and a multitude of states.

Conclusion

Python is good for small programs and rapid software development, but it lacks features I find desirable, like polymorphism, encapsulation or virtual functions. Due to it being an interpreted language, it is also not suitable for applications doing very complex or massive amounts of computations; It is simply too slow. So in the end you need to use libraries like numpy, which wrap C performance into a Python interface. It of course always depends on the application, but for a 3D game – even one as simple as Smiley Battle – you are better off with compiled code. In the end, C++ offers as good as everything Python does and much more. Just use STL and you're good. 😊

In the case of Smiley Battle, Python is good for rapidly developing and testing the application or modules of it and porting it to C++ once everything works.

License

Dietfrid Mali

Contact me at dietfrid.mali@gmail.com

Modules

actor.py

classes: *CActor*

CActor manages most data and functions of an actor in the game. It is the base class for every game element that moves, fires and can collide with something (other actors or walls). It contains methods to create and delete them and to chose attributes for them. It also manages death and respawn states and animations. Classes depending on *CActor* are *CPlayer*, *CViewer*, and *CProjectile*.

actorhandler.py

classes: *CActorHandler*

CActorHandler manages all actors in the game. It contains methods to create and delete them and to chose some basic attributes for them (i.e. pick colors from a pool of available colors). The main data element is an actor list.

arghandler.py

classes: *CArgHandler*

CArgHandler provides functions and data containers for parsing key-value pairs from the command line and from ini files.

camera.py

classes: *CCamera*

CCamera provides functions and data for managing the viewer's and all other actors position and spatial orientation. It basically functions as the ‚eye' of an actor.

collisionhandler.py

classes: *CCollisionHandler*

CCollisionHandler handles all types of collisions (see paragraph Collision Handling). It detects them, computes the results and triggers the corresponding actions.

controlshandler.py

classes: *CControlSpeedData*, *CControlData*, *CJoystick*, *CJoystickHandler*, *CControlsHandler*

CControlsHandler handles all input controls and sets the related game parameters accordingly (movement direction, rotation angle and rotation speed).

CJoyStickHandler handles specifically joystick / gamepad inputs and sets the related game parameters accordingly.

CJoyStick buffers the data of a specific joystick or gamepad attached to the computer.

CControlData manages the game parameters for controlling movement direction, rotation angle and rotation speed and provides methods to manipulate them (e.g. ramp them up the longer the related control is pressed or tilted).

CControlSpeedData holds all data relevant for movement parameter handling. It is basically only a data container without methods.

cubemap.py

classes: *CCubemap*

CCubemap provides methods and data buffers for handling OpenGL cubemaps.

effecthandler.py

classes: *CEffectHandler*

CEffectHandler is a very basic class to provide graphical effects. Right now it can only fade the screen out to or in from black. These effects are used when a player despawns or respawns.

gamedata.py

classes: *CGameData*

CGameData is a container for global game data, like textures or parameters.

gameitems.py

classes: *CGameItems*

CGameItems is a container for the some top level game assets, like map and viewer.

globals.py

classes: None

globals.py contains all global variables of Smiley Battle. It is meant to be directly imported („import globals.py“) to allow for using the global variables and referencing with „globals.<variable name>.

icosphere.py

classes: *CIcoSphere*, *CTriangleIcoSphere*, *CRectangleIcoSphere*

CIcoSphere is a base class for all ico spheres, providing general functionality that is common to them.

CTriangleIcoSphere creates an ico sphere from a triangular base shape (e.g. octahedron).

CRectangleIcoSphere creates an ico sphere from a rectangular base shape (e.g. a cube)

map.py

classes: *CMap*, *CMapData*

CMapData is a container for all map specific data.

CMap provides methods and data for rendering maps, handling actor collisions with map geometry, providing player spawn positions, computing distances and accessing map elements.

maploader.py

classes: *CMapLoader*, *CWall*

CMapLoader provides methods to read game level from file, parse them and construct the internal map data from it. Level data can also be passed in memory, e.g. from the network handler.

CWall is a data container for wall data (vertices, normals). It also provides basic geometric functions required for collision detection.

mapsegments.py

classes: *CSegmentPathNode*, *CSegmentPathEdge*, *CRouteData*, *CSegmentMap*

CSegmentPathNode is a container for data for nodes in a graph connecting map segments.

CSegmentPathEdge is a container for data for edges in a graph connecting map segments.

CRouteData is a container for route data obtained by the DACS based distance calculation.

CSegmentMap offers methods and data containers for map segments. Map segments are the rectangular "cells" of a map defined by the size of a wall. A map segment can be contained by single walls and is the basic building element of game maps. *CSegmentMap* constructs a rectangular map of segments, creates a graph containing LoS (line of sight) connections between the segments and computes the shortest distance between each segment, using the graph's edges and nodes as input to a shortest path search with a DACS based algorithm.

matrix.py

classes: *CMatrix*

CMatrix offers the usual 3D matrix calculation stuff.

mesh.py

classes: *CMesh*

CMesh is a generalized base class for all kinds of meshes (i.e. 3D structures consisting of vertices, faces, color and textures, describing some 3D entity, e.g. a sphere)

networkhandler.py

classes: *CNetworkData*, *CUDP*, *CNetworkHandler*

CNetworkData is a container for game messages received from or to be transmitted via the internet. It basically contains ip address, port and message payload.

CUDP offers basic UPD functionality (opening sockets, receiving and transmitting data).

CNetworkHandler does all the network message sending and processing and implements the control flow for joining or leaving a match, as well as updating all players in regular intervals.

physicshandler.py

classes: CPhysicsHandler

CPhysicsHandler does all the high level physics related actor management, i.e. movement, collision handling and animation. It is part of the main game loop.

plane.py

classes: *CPlane*

CPlane offers functions for fast plane and rectangle related geometry computations (mostly point to plane projections and line – plane – penetrations).

player.py

classes: *CPlayerShadow*, *CPlayerHalo*, *CPlayerOutline*, *CPlayer*, *CViewer*

CPlayer provides data containers and methods to handle player rendering. Everything that is specifically player functionality related (like controlling the life status of a player) is delegated to its base class *CActor*.

projectile.py

class: *CProjectile*

CProjectile manages specifically projectiles with their own requirements to collision handling and feedback and life cycle control

quad.py

classes *CQuad*

CQuad is a container and offers methods for rendering quad based primitives. It can render textures or fill rectangular areas with a color (the latter with alpha blending).

renderer.py

classes: *CRenderer*

CRenderer offers methods for setting up everything around the display and OpenGL as well as providing all required OpenGL projections. It will employ the viewer camera for that purpose. It also sets up the various viewports required by the program.

reticle.py

classes: *CReticle*

CReticle offers methods to load and display the player reticle texture.

router.py

classes: *CDialHeap*, *CRouter*

CDialHeap manages the particular heap data structure required for a DACS based shortest path search. See router.py for details.

CRouter provides an interface for the DACS based distance calculation.

scoreboard.py

classes: *CScoreBoard*

CScoreBoard renders player statuses and scores.

shaders.py

classes: *CShader*, *CShaderHandler*

CShader provides methods for creating and parameterizing OpenGL shader programs.

CShaderHandler contains the code of all shaders uses in Smiley Battle. It creates the OpenGL shader programs.

smileybattle.py

classes: *CApplication*

CApplication is the program's main class. It does all initialization and runs the main game loop.

soundhandler.py

classes: *CSoundObject*, *CSoundHandler*

CSoundObject is a container for sound channel data and provides the link between a pygame.sound sound channel and the sound sources (actors) in the game.

CSoundHandler creates sound sources, links them to their generators (actors) and also releases and unlinks them.

texcoord.py

classes: *CTexCoord*

CTexCoord is a very simply container for OpenGL texture coordinates.

texture.py

classes: *CTexture*

CTexture is a container for 2D OpenGL texture data and provides all functions to load textures from file, register them in the OpenGL context and upload them to video memory.

texturehandler.py

classes: *CTextureHandler*

CTextureHandler is a cross breed between a factory and a tracker. It creates all textures needed in the game, keeps track of them and releases them to OpenGL on program termination. This means that not every game element using textures needs to meticulously keep track of its textures and release them on program termination, simplifying such elements' destructor significantly. It also means that all textures can be released to OpenGL before the program's OpenGL context gets destroyed when the program terminates (which happens before destructors are called, meaning that destructors releasing OpenGL elements cause errors).

timer.py

classes: *CTimer*

CTimer provides some basic functionality for measuring time interval, taking lap times and triggering periodic activities.

vao.py

classes: *CVAO*

CVAO provides a high-level interface to OpenGL's vertex array objects, making registering and describing OpenGL data buffers pretty easy and straight forward. It also offers methods for rendering data attached to them.

vbo.py

classes: *CVBO*

CVBO implements data and methods for OpenGL vertex buffer object handling, making deploying vertex data as VBOs rather simple and straightforward and hiding most of the details.

Vector.py

Classes: *CVector*

CVector provides basic 3D vector math data containers and methods.

Vertexdatabuffers.py

classes: *CVertexDataIterator, CVertexDataBuffer, CVertexBuffer, CTexCoordBuffer, CColorBuffer, CIndexBuffer*

CVertexDataIterator provides an iterator for walking through all available kinds of vertex data buffers.

CVertexDataBuffer is the basic interface class for passing high level Python data to OpenGL's VBOs.

CVertexBuffer is a vertex data buffer for 3D vertices and normals.

CTexCoordBuffer is a vertex data buffer for OpenGL texture coordinates.

CColorBuffer is a vertex data buffer for OpenGL RGBA color values.

CIndexBuffer is an index buffer for indexed OpenGL VBO rendering.