Ceng492 Graduation Project

The Bride Project

Developer Manual

Presented by Meda

Ankara, 2005

Introduction:

This documents aims to give useful information about the implementation of the game. We will quickly start by looking to the files of the Bride and libraries needed and then get into the details of each underlying parts constituting the Bride. We will not however mention the details of the design. Therefore you may want to (indeed we believe you should) look at the “The Bride Final Design Report”, “The Bride User Manual” and “The Bride Coding Standards” before reading this manual. You can find these documents on the web (http://senior.ceng.metu.edu.tr/2005/meda).

File Overview of the Bride:

+ Files related with program entry and driving:
    - Main.h
    - Main.cpp
    - Init.cpp

+ Files related with space:
    * (CVector2 & CVector3)
        - Vector.h
        - Vector.cpp
    * (Vector & Point)
        - utility.h
        - utility.cpp

+ Files related with Camera movement:
    - Camera.h
    - Camera.cpp

+ Files related with ODE (physics engine):
    - physics.h
    - physics.cpp

+ Files related with objects:
    - object.h
    - object.cpp

+ Files related with puzzles:
    - puzzle.h
    - puzzle.cpp

+ Files related with fonts & writings:
    - font.h
    - font.cpp

+ Files related with BSP loading:
    - Quake3Bsp.h
    - Quake3Bsp.cpp

+ Files storing 3d object (model - MD3) related structures:
    - Model.h

+ Files related with 3ds model loading:
- 3ds.h
- 3dsLoader.h
- CJPEGFile.h
- 3ds.cpp
- 3dsLoader.cpp
- CJPEGFile.cpp

+ Files related with images (textures):
- Image.h
- jpeglib.h
- Image.cpp

+ Files related with Md3 model loading:
- Md3.h
- Md3.cpp

Required Libraries:

- opengl32.lib	: graphics engine
- glu32.lib	: graphics engine
- glaux.lib	: graphics engine
- jpeg.lib	: image loader
- winmm.lib	: windows multi media lib
- ode.lib	: physics engine
- openal32.lib	: sound engine
- alut.lib	: sound engine
- python24.lib	: scripting engine

Main Menu:

Ortho Mode:

Ortho mode allows us to use 2D coordinates instead of 3D coordinates. We used Ortho Mode in order to implement the GUI, Inventory and InGameMenu where the textures need to stay in front of the screen and never move. We also have the inverse of the Ortho Mode where we transfer to the Perspective Mode, and to the 3D world again.

Alpha Blending:

In order to have multilayered scene rendering we used alpha-blending to draw GUI, Inventory, In Game Menu, the writings in the game environment and the writings that are printed on the screen in Ortho Mode.

For example: `glBlendFunc(GL_SRC_ALPHA, GL_ONE)` with blending turned on (`glEnable(GL_BLEND)`) will take a texture map and blend it with the background. The closer a color is in the texture map to black, the more it will disappear. If the color being blended is black, it will render the color behind it instead. This means you can have a picture of a tree with a black background and it will only draw the tree and not the black background surrounding it. The problem with this effect is that it makes the tree look transparent / faded. This works fine for particle effects, but if you want the image to be clear and not faded, you will want to either use a 32 bit image with an alpha channel or masking. Masking is basically making it so you only draw the part of an image where its mask image has dark colors. This means you have 2 images, one for the normal picture, and another black and white image that holds the mask. And this is what we did in mapping the InGameMenu.

We again used blending, but with different parameters as:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

where we don't need transparency, like the "GUI", "OPTIONS" menu or the textures mapped to the empty slots in the belt of the hero.

Scene Management:

Frustum Culling:

In order to have efficient scene rendering we culled the faces that are not in our frustum using a technique as frustum culling. Frustum is simply the area of your world visible to your current camera. The shape of the frustum is a pyramid with the nearest peak truncated at what is deemed the 'near' clipping plane. In fact the frustum itself is (or can be) defined by 6 planes. These planes are named the near plane, the far plane, the left plane, the right plane, the top plane and the bottom plane. Frustum culling is simply the process of determining whether or not objects are visible in this area.

Since frustum culling is essentially a 3D world-space procedure, it can be processed long before we even deal with individual polygons in our pipeline. Hence, we can quickly reject on the object level, unlike back-face culling for example which takes place much later in the rendering pipeline and works on a per-polygon basis. This means that we don't even have to send the data to the video card once the object is frustum culled which of course makes quite a difference in rendering speed. It is simply very, very fast to render nothing.

So basically we have pieces of codes that look for if a point, sphere, cube or box is in our frustum. If it is not we have do not have to render those which reduces the heavy weight of the video card a lot.
Thus to implement the technique thinking the frustum as a cube with the center and the half of the length of (like a radius) given, we check all the points if it is in the frustum. If some is found in front of a side we skip that point and pass to another one.

BSP Level:

Actually we imported ready BSP levels as our development levels. These also have their textures with reducing the responsibility of us a bit more.

Characters:

We imported the textures and animations for characters in the game from the tutorials in thewell-known game-developers sites. Actually we are not real artists and it was to hard for us to draw and animate a character on its own, so encouraged by our assitant we used the ready ones.

However, the characters have attributes like health, experience, position which are required for the game engine, and pointers to their polygons (vertex arrays) and to the corresponding textures for the graphics engine.

Implementation of Objects:

The Bride uses Object class to hold all the object information. “object.cpp”, “object.h” contains the code for the implementation of objects appearing in the game. Object class also makes use of “physics.h” where routines of ODE is used for adding physics capability to objects, “utility.h” for point class, “3dsLoder.h” for loading and drawing 3ds objects. 3ds loader have been retrieved from net and is problematic while rendering objects without textures. 3ds models are also retrieved from Internet. Below is the definition of the object class. Each field of the object is commented to show its functionality.

```
class Object
{
private:
        int m_iid;                       // id of the object
        char m_modelfilename[20];        // name of the 3ds model (i.e. chair.3ds)
        Point m_pnposition;              // position of the object
        Vvector m_vcrotation;            // rotation vector defined for that model
        float m_fangle;                  // rotation angle defined for that model
        float m_fscale;                  // scale factor defined for that model
        objName m_ename;                 // enumerator denoting the name of obj. (i.e. box)
        objType m_etype;                 // enumerator denoting the type of obj. (i.e.
                                         //  static)
        objState m_estate;               // enumerator denoting the state of obj. (i.e.
open)
        objAppearance m_eappearance; // enumerator denoting the appearance of obj.
                                         // (i.e. visible)
        objPickability m_epickability;  // enumerator denoting the pickability of obj.
                                         // (i.e. pickable)
        objName m_eusedwith;             // name of the object this oject is used with
        float m_flx;                     // x length for bounding box
        float m_fly;                     // y length for bounding box
        float m_flz;                     // z length for bounding box
        float m_fr;                      // red coloring value if - non-textured object
        float m_fg;                      // green coloring value - if non-textured object
        float m_fb;                      // blue coloring value - if non-textured object
        dBodyID m_dbody;                 // the body id for ODE - if dynamic object
        dGeomID m_dgeometry;             // the geometry representing body – if dynamic
        float hitpoint;                  // Hitpoint of object
        float damage;                    // Damage that object does - if weapon
        C3dsLoader m_3dsLoader;          // 3ds loader object
        float m_fanimationangle;         // value to rotate on VcRotation for animation
        float m_fanimationx;             // value on x to be added to its x for animation
        float m_fanimationy;             // value on y to be added to its y for animation
        float m_fanimationz;             // value on z to be added to its z for animation
public:
	Object();
	Object(Point pnpos);
	~Object();
	inline Point getPosition() { return m_pnposition; }
	inline char * getModelFileName() { return m_modelfilename; }
	inline C3dsLoader get3dsLoader() { return m_3dsLoader; }
	inline objState getState() { return m_estate; }
	inline int getId() { return m_iid; }
	inline void setPosition(Point pnpos) { m_pnposition = pnpos; }
	inline void setModelFileName(char * fname) { sprintf(m_modelfilename, "%s", fname); }
	void setState(objState state);
	void setType(objType type);
	void setAppearance(objAppearance appearance);
	void setPickability(objPickability picktype) { m_epickability = picktype;}
	inline void setId(int id) { m_iid = id; }
	objAppearance getAppearance() { return m_eappearance; }
	objPickability getPickability() { return m_epickability; }
	objType getType() { return m_etype; }
	objName getName() { return m_ename; }
	void render();
	void applyAction(actionType action);
	void addToInventory();
	void takeToHand();
	void use(CVector3 cpos, CVector3 cview);
	void useWith(Object & obj);

	friend int readObjectsFromFile(Object * pobjects, char * filename);
	friend int writeObjectsToFile(Object * pobjects, int nobj, char * filename);
	friend void initializeObjects(Object * pobjects, int nobj);
	friend void renderObjects(Object * pobjects, int nobj);
	friend void setObjectAttributes(Object * pobjects, int nobj, int objid, objType type, objState state, objAppearance appearance, objPickability pickability);
	friend int findNearByObject(Object * pobjects, int nobj, CVector3 cpos, CVector3 cview);
	friend bool checkObjectsForCollision(Object * pobjects, int nobj, CVector3 cpos,  CVector3 cview);
};
```

Though most of the methods above are straight forward to understand, render, setSate class methods and findNearByObject, checkObjectsForCollision routines deserve a little bit more explanation. render method draws the object by looking its appearance and decides where to draw the object by looking if it is visible, in inventory, at hand. If it is visible it takes any possible animation into account before drawing the object in the world. It makes use of 3ds model renderer. setState methods arranges state changes of the object and also calls applyAction if an action defined for that state change. It also sets state change flag as true which alerts a state change and causes updatePuzzle routine of the puzzle module called in the next frame. findNearByObject which is called when 'E' is pressed looks to the position of the model, compares it with the positions and boundaries of objects in object array and returns the index of object in the coverage area if there is any.

checkObjectsForCollision does nearly the same thing but the main difference it is called in camera with possible next position of the character so that it can detect and prevent collisions. It also adds force to the object collided if it is dynamic.

During the initialization of a level, objects of that level are red and initialized from objects file which is by default <level_id>_objects.txt. The objects are stored in a global array called g_pobjects and max number of objects that can be stored is 100. Below is a few lines of the objects file and the comments in the first 2 lines explain what each field denotes.

```
#_This_object_file_need_not_to__be_sorted_with_respect_to_x_and_then_z_position
#Model__objId___Position__RAng__RVec__Scale__OName__OType__OState__O
Appear__OPick__UWith___lx_ly_lz___r_g_b
seating.3ds     1       -1010 -10 -250          0       1 0 0   0.5     8       0       0
        0       1       20      55.0    80.0    15.0    0.3 0.1 0.2
painting.3ds    2       -1280 -10 -55   -90     0 0 1   3.0     7       0       2       0
        1       20      15.0    80.0    60.0    0.0 0.0 0.0
key.3ds         3       -675 410 11700 0        1 0 0   0.06    9       0       2       0
        1       13      10.0    360.0   40.0    0.8 0.8 0.2
```

Implementation of Puzzles:

Puzzle class is used to hold all the puzzle information. “puzzle.cpp”, “puzzle.h” contains the code for the implementation of puzzles. Puzzle class also makes use of “objects.h” for inheriting object attributes. Below is the definition of the puzzle class. Each field of the puzzle is commented to show its functionality.

```
class Puzzle
{
private:
        int m_ipid;             // id of the puzzle
        puzState m_epstate;     // state of the puzzle (i.e. completed)
        puzType m_eptype;       // type of the puzzle (i.e. unordered)
        pelement m_pelements[MAX_ELEMENTS];     // array holding elements
                                // of the puzzle (object id, otype, ostate, oapp., opick.)
        pelement m_pactions[MAX_ACTIONS];   // array holding actions
                                // of the puzzle (object id, otype, ostate, oapp., opick.)
        int m_inelement;        // number of elements
        int m_inaction;         // number of actions occur at the completion of puzzle

public:
        Puzzle(){}
        ~Puzzle() { };
        inline puzState getPuzzleState() { return m_epstate; }
        inline setPuzzleState(puzState state) { m_epstate = state; }

        friend int readPuzzlesFromFile(Puzzle * ppuzzles, char * filename);
        friend int writePuzzlesToFile(Puzzle * ppuzzles, int npuz, char * filename);
        friend void initPuzzles(Puzzle * ppuzzles, int npuz);
        friend void updatePuzzle(Puzzle * ppuzzles, int npuz, Object * pobjects, int
nobj);
};
```

The crucial method in the puzzle class is updatePuzzle which is called whenever state change flag is set, that is at least one object changes state. It checks every element of every puzzle with the corresponding object in the global object array and if all the objects residing in the elements of the puzzle are satisfying the desired attributes the actions defined in the puzzles occurs that is the objects in the action list are set with the new attributes.

During the initialization of a level, like objects, puzzles of that level are red and initialized from puzzles file which is by default <level_id>_puzzles.txt. The puzzles are stored in a global array called g_ppuzzles. Below is a few lines of the objects file and the comments in the first 3 lines explain what each field denotes.

```
#pId__pType__pState__nElms__nActns__dummy__(Elements-are-in-the-following-
line)
#oid__oType__oState__oAppearance__oPickability__(Elements)
#oid__oType__oState__oAppearance__oPickability__(Actions-occur-at-the-end-of-
puzzle:heroid=-1,headassasinid=-2)
1       0       0       1       3       #
12      0       0       0       1
7       0       1       0       1
8       0       1       0       1
9       0       1       0       1
2       0       0       3       1       #
7       0       0       0       1
8       0       1       0       1
9       0       0       0       1
2       0       1       0       1
```

Implementation of User-Hero-Object-Puzzle Interaction:

Implementation of user-hero-object-puzzle interaction is the heart of the game. Main.cpp is the program entry point. In this code in a loop inputs are checked, puzzles are controlled and then objects and characters are rendered. Camera, object, puzzle, inventory, Quake3Bsp classes are used in the main.cpp to handle the movement and selection requests and call necessary routines for puzzles and objects (i.e. updatePuzzle, findNearByObject) and change their attributes. In addition to these classes font, menu, physics, sound, utility, vector implementations constitutes the whole game. The scene writings and system messages use “font.cpp”, “font.h”.

Contact:

Website: http://senior.ceng.metu.edu.tr/2005/Meda

You may visit our website for more information about the game and contact information. Also don't forget to check it for new updates.

Since it is possible that this page can be removed from server, please do not hesitate to contact via email to emreguney@gmail.com or ibrahim_aliz@yahoo.com for any questions and comments.