Student Name: Militaru Cristian
Group: 30431

– Graphical Processing Systems project –

Table of Contents

1. Subject Specification

OGEL is a virtual Lego© game. There is a large board on which the player can place pieces and build any non-moving object.

2. Scenario

2.1. Scene and objects description

The scene contains initially only the base board, on which the player can place pieces. During the game-play, the number of pieces grows. There is a light source above the board that provides illumination to the scene. When the player is in view mode (explained later), there is also a light situated in the position of the camera, enabling the player to see better the objects around him.

2.2. Functionalities

- Camera movement: the camera can be rotated around the center of the screen, enabling the player to see his creation from different angles. Also, the user can zoom in and out.
- View mode: the camera can be moved freely in the 3D space.
- Object manipulation: the player can choose what type of piece to place, then move it to the desired location, and finally to place it above the other piece.

3. Implementation details

3.1. Functions and special algorithms

Special algorithms:

- camera movement: moving the camera around the scene.
- checking if two pieces intersect on the xOz plane: required to check if a piece is situated above another piece.
- color choosing: allows changing the color of the current piece.
- piece rotation: the user is able to rotate the current piece, creating a “new” piece shape.

3.1.1. Possible solutions

Camera movement:

This problem can be solved in many ways. Two of them are:

- We can consider that the camera movement to the left is equivalent to moving all the objects in the scene to the right. This can be accomplished using a translation on the projection matrix. Rotations can be implemented in a similar way. This method requires the camera coordinates and the angles of the camera orientation.
- Using the gluLookAt function. This requires knowledge of the camera coordinates and the camera direction vector.

In both cases, the camera coordinates are known in a free-space camera (when movement is accomplished using the keyboard arrows). The direction can be computed using the sine and cosine functions on the rotation angles. The rotation angles are modified by the user when moving the mouse cursor.

Collision checking:

Checking that two pieces collide is not an easy task. Two methods that I've thought of are:

- Using many conditions based on the two pieces' coordinates and their sizes.
- Using matrices to represent their projections on the 2D xOz plane.

Color generation:

If the pieces can only have values from a small set of colors, we can define those colors as constants and simply provide one of them when requested.

Piece rotation:

In order to rotate a piece, we can attempt multiple approaches:

- For each shape of piece, provide 4 shapes in the implementation, one for each possible rotation (0, 90, 180, 270 degrees). This is problematic due to code repetition.

– When we rotate a piece, we interchange their sizes on Ox and Oz axis, and draw them with a rotation. This is a valid choice since all the pieces have the exactly same behavior, no matter how they are rotated, and only the visible part is modified.

3.1.2. The motivation of the chosen approach

For camera movement, I chose to use the gluLookAt implementation because of familiarity with the method.

I implemented the collision checking using many checks for simplicity reasons and also because it requires no additional memory.

The color generation fits the problem best since we only have a few possible colors.

The piece rotation uses the simpler algorithm in order to keep the code shorter, simpler and less redundant.

3.2. Graphics model

The pieces that are on the scene are drawn only using basic OpenGL and GLU shapes. That is, no 3D models were actually used. This provides a speedup since the file readings (which would otherwise provide no advantage) are avoided. The used shapes are:

– polygons: for the pieces.

– circles and cylinders: for the pieces' connectors.

All the pieces have a rectangular base.

Some of them do not have a cuboid shape. They can have slopes. These are the „edge” and „corner” shapes.

A difficult problem in this case is finding the normals of each surface in the scene. They were computed manually, but after that I thought that the problem could have been solved automatically (the normals could have been computed by the computer with ease).

3.3. Data structures

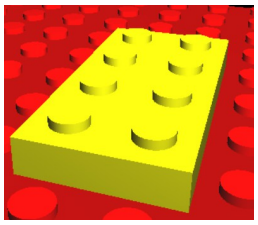
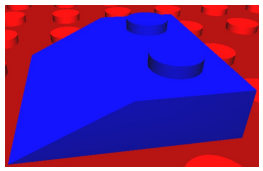
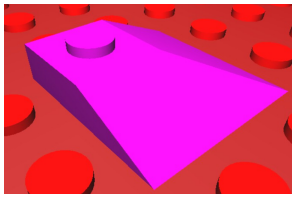
The scene is composed of pieces. There is a constraint when adding a new piece: it must be placed on top of another piece. For that reason, we need to have all the pieces stored in one place so we can iterate and check this constraint.

To implement this, I used a vector from the standard library, which contains all the pieces that are in the scene.

3.4. Class hierarchy

The backbone of the application is the RectangularPiece class. It models the common attributes and behavior of all the pieces. The attributes are: the position, the size and the color. A piece must be able to rotate itself and to draw itself on the screen (these are actually abstract methods, since each shape might be drawn differently).

The RectangularPiece class is extended by all of the possible shapes:

- SimplePiece 
- EdgePiece 
- CornerPiece 

The ColorGenerator class is used to enable multiple coloring of the pieces. It returns a color from a predefined set, each time it is requested.

The PiecesContainer class holds all the pieces in the scene. It also contains the collision checking algorithm.

The PieceFactory class is used to provide new pieces when requested. That is, when a piece is placed, a new piece is provided. This new piece can have any shape of those described above.

4. Graphical user interface

The user interacts with the system using the keyboard and the mouse. In this game, there are two interface modes:

- build mode: the hotkeys are set for manipulating the objects:
  - the left, right, up, down arrows: move the current piece (that is going to be placed).
  - R: rotate the current piece.
  - T: change the type (shape) of the current piece.
  - Spacebar: place the current piece above the piece below it.
  - C: change the color of the current piece.
  - + and – : camera zoom in / out.
  - V : switch to view mode.
  - Z : undo the last piece placed.
- view mode: move freely around the scene:
  - The left, right, up, down arrows: move the camera.
  - V: switch to build mode.

In both modes the user can use the keys:

- W: for wireframe display.
- Q: quit the application.

The mouse is used as follows:

- In build mode: hold the left mouse button down while dragging, in order to rotate the camera about the center of the scene.
- In view mode: moving the mouse rotates the camera around its position.

5. Conclusions and further development

The game is a simplistic approach to a „virtual Lego”. It has many features that enable the user to create a small world

The game can be extended in many ways. One of them would be to enable the player to select any piece in the scene with the mouse cursor and remove it. Or a save & load game feature would be very cool. Furthermore, adding more complex pieces such as trees, doors or post lamps would greatly improve the complexity of the game.

6. References

I used a little of the code from the laboratories and Stack Overflow in dark times.