S

Introduction

Project STORM is a video game. A video game is a virtual reality that is only partially controlled by the user and mostly governed by a set of laws that describe what the user may do and how objects and aliens react in this universe. In our case, the laws are written in the C programming language and the reality is perceived as a 2D projection on a video screen, sound output from speakers or headphones.

Although at first glance the game may give you the impression of a 3D universe, most of the action can be described using a two-dimensional construct. Imagine a rectangular piece of transparency floating in 3-space. Now divide this rectangle into **n** evenly spaced lanes. If the lane boundaries are drawn with color, you will see what in Project STORM terminology is called a field. In addition to the flat field that we now have, the rectangle can be bent to different shapes to confuse the player and have a variable angles between different lanes.

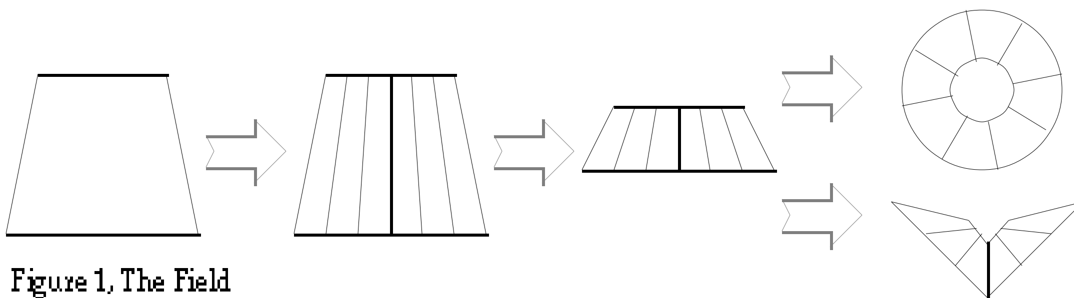
Figure 1, The Field

If you further divide the field into a discrete number of slices going from the screen to the end of the field, you get something we call the grid. The grid is divided finely enough to make movement look smooth, but coarsely enough to save memory requirements, since every point in the grid is precalculated. The grid structure (called ww) is the most important global variable in the game. It contains screen coordinates for for all grid points, some points in between grid points and even some important points outside the field.

Before we define how the player, aliens and objects are allowed to move, it might be useful to give an idea of how game play proceeds. It is quite easy to do this in the form of a flowchart:

Before the actual game begins, the player is allowed to choose from a number of different starting levels to match difficulty and skill.

While the screen shows a classic starflight effect (à la Star Trek®), grid points are calculated and stored. The calculations take into account the size of screen that is used. Finally, the field is approached from a distance.

Game play is actually your basic shoot’em’up with some twists added to make the game interesting and visually pleasing. Game rules are described in another document.

Assuming that the player survives a level, clearing all the aliens assigned to that level, he/she still needs to fly through the field.

Lanes may be obstructed by spikes, so the player either has to destroy or avoid the spikes. Hitting a spike will reduce one life from the player and return him/her to the edge of the field. This goes on until the player either successfully completes the level or looses all lives.

Assuming that the player has survived as far as this, he/she is awarded bonus points, if this is the first level. A special weapon is recharged and a new level is loaded from disk.

When the game finally ends, player score is compared to a high score list, and if the score is high enough, a dazzling fireworks display is awarded along with a chance to enter initials in the high score list.

Game Laws

There are some laws that constrain the movements of all objects in the game. Each type of object has its own set of rules. It is easier to understand these rules, if you imagine the grid as a canyon or well. One could say that there are nasty things **down** there climbing **up** to the **edge**.

The actual coordinate system measures the distance from the edge, so the edge has a depth coordinate of 0. The edge is assumed to lie on the screen and so most of the game action happens behind the screen. The deepest visible point of the grid has a depth coordinate of DEPTH, where DEPTH is a constant defined in the file storm.h. The distance of the screen from the user is contained in the constant STARTDEPTH. These constants are sufficient for the programmer to draw a perspective view of the grid and for the reader to understand the laws of the game.

The Player

The player is a claw-like creature that can usually only move along the edge of the grid. The only times the claw is free to move from the edge is when the player completes a level or when the player is captured and dropped down from the edge. The claw occupies one lane at a time. To give the impression of free movement, the claw twists when the player moves it without changing lanes. For a better idea of how the player moves, try out the game or a prototype of the game.

Player shots move down the lane that the claw occupies. Shots will explode when they hit something. If a shot goes down to the bottom, reaching DEPTH, it will just disappear without causing any damage. The number of simultaneously existing player shots is limited to make game play more interesting.

Flippers

Flippers do two things simultaneously. They move up from the grid, trying to reach the edge. While doing this (and while on the edge), they rotate. To understand how flippers rotate, imagine a gymnast doing cartwheels. If a flipper reaches the edge and captures the claw, the claw will drop from the edge and the player will loose a life.

Spikes and Spikers

Spikers move only up and down the lanes and while doing so create spikes. Spikes always start from the bottom of the grid and reach up. Initially, all spikes are the same size. A spiker will appear from the bottom of the grid, move up until it reaches a certain level and then move back down. While it is moving up, it builds the spike like a spider moving along a thread.

Spikes can be shortened by hitting them with a shot. The main function of spikes is to act as an obstruction between the player and the enemies. To defend itself from random shots, a spiker will usually fire a plasma ball for protection. The plasma ball will detonate a single shot coming down the lane and is also capable of destroying the claw.

Fuseballs

A fuseball usually does only brownian motion randomly on the grid. It may sometimes decide to quickly leap upwards trying to scare the player. It often reaches as high as the edge and once there, it will not descend again. If the claw touches a fuseball, it will disintegrate, costing the player a single life.

Pulsars

Pulsars are a lot like flippers, but they move and act slightly differently. When a pulsar lies on a lane (having flipped there from another lane), it “pulses”. Pulsing makes the pulsar shape flatten and at the same time causes a void on the edge. If the claw is located on a lane with a void or moves onto one, the claw is destroyed and the player looses a life.

Tankers

There are three types of tankers: flipper tankers, fuseball tankers and pulsar tankers. Tankers move along a lane towards the edge. If a tanker is hit or it reaches the edge, it will break into two flippers, fuseballs or pulsars depending on the type of tanker.

Drawing Shapes

A global structure called “grid” contains grid coordinates and width of a lane at every depth that is used in the game. Here’s the latest version of the grid variable. The instance of this variable is called ww (a short name was chosen. w is shorthand for world).

```
typedef      struct
{            int                   numsegs;              /*           Number
of segments on this grid.  */
             int                   wraps;                             /*
             Does this grid wrap around?                              */
             int                   *unitlen;             /*
             Pointer to table of segment widths.     */
             int                   *x[MAXSEGS];  /*           Tables  of  segment
edge x coords.           */
             int                   *y[MAXSEGS];  /*           Tables  of  segment
edge y coords.           */
             int                   *xc[MAXSEGS]; /*           Tables  of  segment
center x coords.         */
             int                   *yc[MAXSEGS]; /*           Tables  of  segment
center y coords.         */
             int                   starsegs;             /*           Number
of divisions for stars.            */
             int                   starx[STARDIVISION*MAXSEGS][2];          /*
             center star x stuff       */
             int                   stary[STARDIVISION*MAXSEGS][2];          /*
             center star y stuff       */
}            GridWorld;
```

Segment numbers start from 0 and end at ww.numsegs. To put a pixel on spike top on lane 0 and depth level 10, you could write the following:

```
VA.color=0;
VAPixel(ww.x[0][10],ww.y[0][10]);
```

As you can see, the way this structure is defined makes it quite easy to draw within the grided coordinate system.

Legal depth values range between ZOOMDEPTH and DEPTH (including these values). ZOOMDEPTH is typically negative (it’s behind the player) and represents the distance of the screen from the user. DEPTH is the distance from the screen (or top of the grid) to the bottom of the grid. Only values between 0 and DEPTH are guaranteed to be on the display. All other values should go through range checks or clipping routines before they are used for drawing. There’s a guaranteed margin of ww.segmscale[0] between the edge of the display screen and the grid.

Sometimes staying within the grid is not desirable. Think about the flipper shapes and how they are drawn. Only one edge of the flipper is connected to the grid while the flipper is flipping from one lane to another. In order to get this effect, the unitlen array is used. To draw a line that is connected to the counterclockwise edge of lane 0 on level 10, you start out with the following:

```
x = ww.x[0][10];
y = ww.y[0][10];
VA.color = 0;
```

```
VAMoveTo(x,y);
```

To draw a single segment wide line at an angle of 60 degrees, you can do the following:

```
angle = 60 * ANGLES / 360;
x += (Cosins[angle] * (long) ww.unitlen[10]) >> 8;
y += (Sins[angle] * (long) ww.unitlen[10]) >> 8;
VASafeLineTo(x,y);
```

Note that the Cosins and Sins arrays are scaled integers, where the value 1.0 is represented with 256. Shifting by eight bits is a close approximation of dividing with 256 even with negative values, where there is a small error.

There are two more global variables that you should be aware of when drawing shapes. These contain the angles at which the lanes are on the screen. You only need these arrays when what you are animating should flip from one lane to another. The arrays are called PrevSeg and NextSeg. Read the source file STFlipper.c and find the function UpdateActiveFlipper for some commented sample code on how to use these values.

Creating New Enemies

The main interface for creating a new STORM enemy is kept in the STInterface.c source file. There are three routines that most enemies should install.

The **Alloc** routine is called at program startup. The alloc routine should be named after the enemy type you are creating and it should allocate any dynamic storage that your enemy will need during game play. No way has been provided to deallocate storage, but usually that is not a problem. Here's a typical allocation routine:

```
void		AllocFlippers()
{
		Flippers=(Flipper *)NewPtr(sizeof(Flipper)*MAXFLIPPERS);
}
```

The **Init** routine is called at the beginning of a new level and whenever a level is restarted after the player has lost a life. Usually you only need to zero your active object count (the number of active enemies of that type). Here's a typical initialization routine:

```
void		InitFlippers()
{
		register	int			i;

		for(i=0;i<MAXFLIPPERS;i++)
		{		Flippers[i].state=inactive;
		}
		ActiveFlippers=0;
```

```
}
```

The third routine does most of the work and all the accounting. It relies heavily on the ThisLevel-variable structure. You can add your own fields to this structure by changing the ststruct.h include file and the stresource.c source file. You are free to modify your own parts of this structure. For instance, every time one of your enemy objects is destroyed, you can subtract one from the active count as follows:

```
ThisLevel.flCount--;
ThisLevel.totalCount--;
```

Always remember to update the totalCount as well as your own counter [the flipper count was used a sample here, you should use your own counter name]. When ThisLevel.totalCount reaches the same value as ThisLevel.edgeCount, the game level end counter is started. So, if your enemies reach the top edge of the grid, you should add one to ThisLevel.edgeCount. Since ThisLevel.edgeCount is zeroed every time through the loop, you should increment it every time your update routine is called.

The update routines should also update the amount of stars that are still needed and the total number of active enemies:

```
ThisLevel.starCount += ThisLevel.flCount-ActiveFlippers;
ThisLevel.activeCount += ActiveFlippers;
```

The activeCount field is used to determine how busy the player is. It is compared with the boredomCount field to determine if new enemies should be created more often.

Since update routines can be long and complicated, you should examine existing code to see how things work. STFuseBall.c is probably the easiest one to understand, so start with that and then look at the others, if you wish to create your own enemy types.

The biggest problem with adding new enemy types is that the editor that is currently being developed does not support new structures. You can either create your own resource types for your own enemies or you can opt to change the editor. Hopefully the editor will some day be upgraded to handle easy addition of new enemy types.

Note that the full source code for STORM is expected from anyone attempting to add new enemy types. Anything less will make the task quite hard.

---------- Footnotes ----------

1. Spikes exist in the middle of lanes, so it is convenient to store them for faster and easier access.
2. Before aliens are visible in the field, they are represented by stars. Stars seem to move freely near the center of the field shape, but they are actually constrained by the shape of the field.
3. The SuperZapper
4. To make a level easy, make the initial lengths of spikes 0, by settings the depths to DEPTH+1.
5. All measurements in the grid structure are in screen coordinates.

---------- Sidebars ----------

Created: Sunday, October 28, 1990 Last change: Wednesday, April 24, 1991

Project STORM

Author: Juri Munkki

Basic Concepts

Copyright ©1990, Project STORM Team

To understand how Project STORM works, it is necessary to cover some basics on the “physics” of the STORM universe. Some data structures are also explained.