Procedural Textures System Design Document, v2

Randy Angle

October 19, 1999

Introduction

Provides animation without incurring additional art assets. This system allows for visual effects involving procedural textures using simple 2D algorithms over time. The sprites will be updated regularly (each frame if necessary) and can be copied to a location on the screen to represent a special effect. Think of this a tricky color cycling or Budweiser Beer sign animation (a scrolling lit background behind a transparent picture of a river). The algorithm will have several dial-able properties and can change over time. The other purpose of the procedural texture system will be to provide wipes and fades used during screen transitions.

Requirements

A procedural texture creates a new instance of a TexFX class, which is a custom form of the CAnimSprite. It will use parametric values that change over time to display a unique animating rectangle sprite object. It must be able to sustain the animating of the sprite in a real-time performance using simple methods:

```
pWaterSprite = new CTexFX;
pWaterSprite->Spawn(ETFXSTYLE_WATERFALL,Speed,Xpos,Ypos,Size,Layer);
pWaterSprite->NewOrigin(x+dx,y+dy);
pWaterSprite->NewSize(Box);
pWaterSprite->NewStyle(ETFXTYLE_FIRE);
pWaterSprite->NewSpeed(Speed+5);
pWaterSprite->NewLayer(Layer);
pWaterSprite->Update();
pWaterSprite->Draw();
delete pWaterSprite;
```

Procedural textures are the same as any other animated sprite object and can be displayed in any layer. The larger they are the slower they become because of the number of pixels that must be read-modify-write to make the procedure work.

Because they are simple rectangles it may be necessary to place them behind other objects that define non-rectangular shapes.

This module uses the graphics system. While very similar to the CAnimSprite class, it may have features that would make it unique. TexFX objects are attached to the Scene database as they are created.

Structures/Classes

```
typedef enum
{
        TFXSTYLE_WATERFALL=0,
        TFXSTYLE_FIRE,
        TFXSTYLE_LAVA,
        TFXSTYLE_BUBBLES,
        TFXSTYLE_STATIC
} ETFXSTYLE;

class CTexFX : CSprite
{
private:
        int            Speed;
        ETFXSTYLE      TexFXStyle;
        CBitMap*       pOldImage;

public:
        CTexFX();
        ~CTexFX();
        Spawn(  ETFXSTYLE TexStyle,
                int Speed,
                long Xpos,
                long Ypos,
                RECT Size
                int Layer);
        CBitMap* Attach(ETFXSTYLE TexStyle, CBitMap* pScreen); // returns old bitmap pointer
        void    NewStyle(ETFXSTYLE Style) {TexFXStyle = Style;};
        void    NewSpeed(int newSpeed) {Speed = newSpeed;};
        void    Update(void);
}
```

Schedule Task List

System Tasks	Duration	Dependent
Design TexFX System	1 Day	Design Document
Code TexFX System	3 Days	TexFX Class designed
Integrate TexFX System	1 Day	TexFX Class coded
Test & Revise TexFX System	1 Day	TexFX System integrated
Rework #1 TexFX Class	1 Day	As Needed
Test & Revise TexFX Rework #1	1 Day	TexFX Class Reworked #1
Rework #2 TexFX Class	1 Day	As Needed
Test & Revise TexFX Rework #2	1 Day	TexFX Class Reworked #2
Total	10 Days	

Memory

The TexFX system uses a bitmap that is loaded into system RAM for sprite data. The only other memory is the instance of the TexFX class. This should amount to less than 1K of RAM per TexFX class and width*height*2 per bitmap.

Risk Assessment

The real risk with the TexFX system is that it could be very slow to have multiple TexFX sprites on screen at once. Every effort will be made to optimize the performance so that it is possible to have at least three TexFX sprites on screen at once. If necessary the update rate can be slowed dynamically to maintain frame rate.

QA & Test

The TexFX system relies on much of the planned sprite technology. All the QA department should concern themselves with is does the presence of TexFX sprites slow the game too much and is the correct effect displayed.