Introduction

The vector animation toolkit requires an 8-bit display and at least a 68020 processor. Animated double-buffered drawing can be done using display lists for different types of objects. Lines, simple points and large points comprise the set of basic animation operators. The kit also allows more complicated objects such as explosions. A separate subroutine allows fast drawing of 7-segmented displays for displaying numbers.

The 8-bit display is divided into several bitplanes. There are static and animating bitplanes. Both types have their advantages and both are used in Project STORM. Project STORM uses two single-bit static bitplanes and two 3-bit animating bitplanes. The current implementation allows only one pair of animated bitplanes for double buffering.

To give you a better feel of what can be done with the vector animation toolkit, here's a short program that uses the most basic parts of the kit:

```
#include "VA.h"
#define      N            3

void         main()
{
             int                                                  i;
             Point                                  vector[N];
             Point                                  speed[N];

             DoInits();
             GetNewWindow(1000,0,-1);
             randSeed=TickCount();

             VAInit(GetMainDevice());

             for(i=0;i<N;i++)
             {            vector[i].h=((unsigned int)Random()) % VA.frame.right;
                          vector[i].v=((unsigned int)Random()) % VA.frame.bottom;
                          speed[i].h=1;
                          speed[i].v=1;
             }

             do
             {            VA.color=255-128;
                          VA.segmscale=20;
```

```
                        VADrawNumber(TickCount(),200,VA.segmscale+VA.segmscale+5);

                        VA.color=3;

                        for(i=0;i<N;i++)
                        {           vector[i].h+=speed[i].h;
                                    if(vector[i].h<VA.frame.left                ||
vector[i].h>=VA.frame.right)
                                    {           vector[i].h-=speed[i].h;
                                                speed[i].h=-speed[i].h;

            VAExplosion(vector[i].h,vector[i].v,3,2);
                                    }
                                    vector[i].v+=speed[i].v;
                                    if(vector[i].v<VA.frame.top                 ||
vector[i].v>=VA.frame.bottom)
                                    {           vector[i].v-=speed[i].v;
                                                speed[i].v=-speed[i].v;

            VAExplosion(vector[i].h,vector[i].v,3,2);
                                    }
                                    if((Random() & 255)==0)
                                    {
            VAExplosion(vector[i].h,vector[i].v,2,3);
                                                speed[i].h+=speed[0].h;
                                                speed[i].v+=speed[0].v;
                                    }

                                    VASafeSpot(vector[i].h,vector[i].v-1);
                        }
                        VA.color=0;
                        VAMoveTo(vector[N-1].h,vector[N-1].v);
                        for(i=0;i<N;i++)
                        {           VALineTo(vector[i].h,vector[i].v);
                        }

                        VAStep();
            } while(!Button());

            VAClose();
}
```

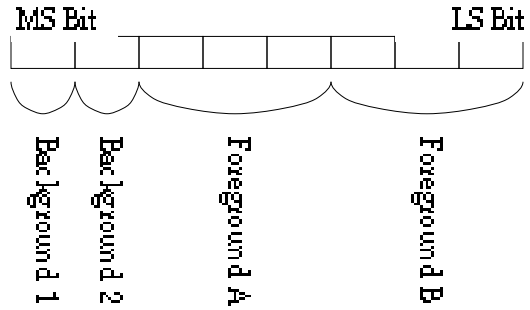
The program draws a polygon of N sides and bounces it on the screen. Collisions with the borders create explosions and at random times the speed of movement changes. Almost all the basic elements of the toolkit are used. Line segments are used to draw the polygon sides; spots (large points) are used to point the corners of the polygon; a 7-segmend display shows the global TickCount of the system (a timing variable) and explosions demonstrate the single pixel objects.

Please note that the above program only supports a single screen. To support selection from multiple screens, use the ScreenSelect function found in ScreenSelect.c. The screen selection code was not made an integral part of the vector animation toolkit because it also displays a dialog that is closely related to the STORM game itself.

Color Selection with the Toolkit

The vector animation toolkit uses an indexed color selection scheme. This means that you can't use the normal QuickDraw RGBForeColor and RGBBackColor calls with it. To understand how colors are used, you have to understand how the 8 bitplanes are divided. In this chapter we will assume that the standard STORM bitplane division is used. In many cases you might prefer using a simpler division with more colors available, but no provisions for background graphics.

Every pixel on the screen is made up of 8 bits. Usually these represent 256 distinct colors, but in our case, we create a special color table that creates bitplanes:



Foregrounds A and B are never displayed at the same time. The toolkit color mapping always makes one of these transparent, so that double buffered drawing can be done invisibly to the user. Usually you can assume that you have one Foreground plane (3 bits wide) and two background fields. If the visible foreground field is transparent (color index 7), background 1 is displayed unless it is also transparent (color index 1), in which case background 2 is displayed unless it too is transparent (color index 1), in which case the pixel has the background color. The background color is usually black, but you can create some interesting special effects by changing it to some bright color for an instant (blinking the screen).

Color drawing uses three variables to set the bitplane and color values. VA.color is the value that is used for drawing. The width (in bits) of this value is stored in VA.field. Foreground fields have a width of 3 and background fields are one bit wide. To indicate what field to draw in, you have to set the VA.offset field. Here's a table of useful VA.offset and VA.field values:

	VA.offset	VA.field
Background 1	0	1
Background 2	1	1
Foreground A	2	3
Foreground B	5	3

You have ten colors available. These colors are usually read in from a 'CLOT' resource. Each color is represented by 8 bytes in the resource. The first two bytes are currently unused by the toolkit and the rest are

16-bit values for red, green and blue components. The first seven values are forground colors, then there's background 1, background 2 and the screen background color.
All commands that use the double buffered display lists only need the VA.color value, because VAStep that does the actual drawing sets these values for you. You have 7 colors and transparent available for drawing.

Most of the time you only need to touch VA.color to draw. You never need to erase anything you have drawn with display list commands, because everything that is drawn with display lists is erased before the next frame is drawn. For this reason, you usually only use VA.color values between 0 and 6. What these colors are, depends totally on what you have set them to.

If you wish to modify the background graphics, you have to set the offset and field variables and call the static drawing commands. Our sample program doesn't have any static background graphics, so it doesn't change the values of VA.field and VA.offset. Currently there are only two static line drawing commands: VAStaticLine and VADrawNumber. You can do some static drawing with color QuickDraw.

Color QuickDraw doesn't understand our concept of bitplanes. Instead, it draws either with RGB values or color index values. In our case, the RGB values have no use, since the color table has the same color repeated in many places and this confuses the color lookup routines.

Indexed mode has the problem of affecting all color planes at once. To activate QuickDraw drawing, thePort has to be VA.window. To select the color to be used, you call PmForeColor with an 8-bit value that represents the value you want to write to the pixel. To draw on background 2, color index 191 is used.

At one point, the animation toolkit had the following lines in the routine that erased the screen. They wrote a short copyright notice in the bottom left corner of the display:

```
            SetPort(VA.window);
            PmForeColor(BG2);
            MoveTo(20,VA.frame.bottom-20);
            DrawString("\PProject S.T.O.R.M. Version 0.7, ©1990 Project STORM
Team");
```

To provide more flexibility, I implemented another way to select indexed colors with color quickdraw. This involves a special color matching routine that is attached to the display device. Since the palette manager provides enough flexibility in most cases, you should read the souce in VAColor.c, if you need to use this special color scheme. If you ar e drawing a quickdraw PICTure, you might find it useful to be able to change the color matching code.

If the way colors and transparency is handled doesn't please you, you can

replace the internal routine VASetColors with your own version. This routine precalculates all the logic needed for the bitplane arrangement needed for Project STORM. If you find you need 16 discrete colors and no static background graphics for a tank game, change this routine and the field and offset values in VADoFrame and you are all set with a different color mapping scheme!

Using the Vector Animation Toolkit

This section describes vector animation toolkit routines and how they are used. In addition, there is the VAMoveTo preprocessor macro that emulates a very simple subroutine call. It is described as a subroutine although it actually isn't one.

Vector Animation Toolkit Calls

void VAInit(GDHandle device);
void VAClose(void);
void VAEraseBuffer(void);

Call VAInit only once when you start up your application. You select the monitor you want to use by passing a graphics device handle. If you only wish to use the main screen, use VAInit(GetMainDevice()) and everything should be fine. For publicly distributed programs, you may want to use some screen selection code. While not a part of the vector animation kit, Project STORM has a program that puts a window on every screen and allows the user to choose a screen with a mouse click.

VAClose doesn't currently properly deallocate storage, so you can't really totally close the kit. To temporarily use normal toolbox graphics, hide the VA.window and call VAClose to restore the old color map. To restart, you need to show VA.window, reset its visRgn to contain the whole display, call VAEraseBuffer to erase the display and then VAStep to restore the VA color mappings.

void VASetColors(Handle theColors);

Use VASetColors to set the ten colors that are used on the screen. For more information on the format of the parameter, read the previous section. You can call this routine at any time after VAInit, but it the color changes will only take effect when you call VAStep. If you only need color animation with no "real" animation, take a look at how VAStep call SetEntries and do it yourself to change just the color table of the display.

Routines That Use Display Lists

void VALine(int x1, int y1, int x2, int y2);
void VAMoveTo(int x, int y);
void VALineTo(int x, int y);
void VASafeLineTo(int x, int y);

These are the four display list line drawing commands. VALine takes four parameters and draws a line from (x1,y1) to (x2,y2). VAMoveTo is in reality a macro and it simply moves the current point to coordinates (x,y) without issuing any drawing commands. To draw lines, call VALineTo or VASafeLineTo.

Unless you use the "VASafeLineTo" call, you have to make certain that you are not drawing outside the displya frame (in VA.frame). Drawing routines do not do any clipping. VASafeLineTo doesn't do any real clipping either: it just discards any lines that are outside the frame. Use VAClip to clip, if you absolutely must, but remember that clipping can be costly and in most cases VASafeLineTo is sufficient.

Use VA.color to set the color of the lines to be drawn. VA.offset and VA.field have no to animating commands such as these.

void VAInitFractalLines();
void VACloseFractalLines();
void VAFractalLineTo(int x, int y, int factor, int level);
void VAFractalLine(int x1, int y1, int x2, int y2, int factor, int level);

Fractal lines should be used with care, since they produce a number of normal lines (2^level, to be exact). You should open the fractal line package before using it and you should close it when you are done. (You can open and close the package any number of times, but only the first open and last close will be effective.)

The factor argument of the line determines how wrinkled the line appears. A smaller number will produce more wrinkled lines. Use 128 as an initial guess and change it to suit your needs.

void VAPixel(int x,int y);
void VASafePixel(int x, int y);

These routines draw animated single pixels and have no effect on line drawing (they do NOT do a VAMoveTo).

void VASpot(int x,int y);
void VASafeSpot(int x,int y);

Spots are groups of 5 pixels arranged like this:

Otherwise spots work just the way pixels do.

void VAExplosion(int x, int y, int size, int color);

Call VAExplosion to start a fireworks-like explosion at (x,y). Drawing is clipped to the screen frame, so you are safe to call this routine with any (x,y). There are currently 4 different sizes of explosions numbered from 0 to 3. The standard explosions are read from a set of resources. New explosions can be created by modifying the supplied source code in Explosion.c.

void VAStep(void);

VAStep does all the hard work. If it has time, it first draws new graphics on the hidden foreground buffer, displays this buffer by calling SetEntries and then erases graphics on the now hidden screen. Think of this routine as the analog of Flush(). You need to call it every time you start a new animation “tick”. VAStep goes into a waiting loop if it detects the animation is running too quickly. In most cases it does its job and returns immediately.

You can change the pace of the animation by changing VA.FrameSpeed. The time allocated for one frame is VA.FrameSpeed/60 seconds. Usually 20 frames per second is quite sufficient, so a default value of 3 is initialized in VAInit.

void VACatchUp(void);

VACatchUp can be called after a time-consuming operation when you wish to continue with smooth animation. This routine resets all timers to such values that the next frame will be drawn and the animation system doesn’t consider itself being “late”. It doesn’t do any drawing.

Background Graphics

The following two calls are used for background graphics. They do not affect the display lists and draw immediately before drawing. To set the color you wish to draw in, you have to set VA.color, VA.offset and VA.field to correct values.

void VADrawNumber(long num, int x, int y);

VADrawNumber draws a long integer as a 7-segment style number. The coordinates are for the bottom left corner of the rightmost digit. The height of the number is 2*VA.segmscale+3 and a single digit is VA.segmscale+5 wide. The only clipping is agains the left border of the display. You have to be careful not to draw above, below or right of the display frame.

void VADrawText(char *text, int start, int length);

VADrawText is analoguous with the QuickDraw call DrawText, but it only allows you to use the vector animation kit 14-segment font. (Upper case, numbers and a few special signs are available.) The position can be controlled with VAMoveTo and text height is VA.segmscale * 4 + 4 and width is VA.segmscale * 3 + 3.

void VAStaticLine(int x1, int y1, int x2, int y2);
void VAStaticLineTo(int x, int y);

Call VAStaticLine to make more permanent changes to the display. No clipping is done and drawing is performed immediately. VAStaticLineTo can be used along with VAMoveTo to draw polylines.

Miscellaneous Routines

int VAClip(Rect *ptrec);

VAClip can be used to clip lines. Put the coordinates of the line to be clipped into (ptrect.left,ptrect.top) and (ptrect.right,ptrect.bottom) and call VAClip. -1 is returned if the line is visible, 0 if it is outside the frame. The clipped line is returned in the same rectangle structure, but there are no guarantees that the endpoints haven't been swapped.

Further Reading

To fully understand the vector animation toolkit, you should be familiar with display lists, double buffering and quickdraw graphics. More information can be found in:

"Principles of Interactive Computer Graphics", Newman+Sproull, McGrawHill
"Graphics Gems", A. Glassner, Academic Press
"Inside Macintosh I", Apple Computer Inc., Addison Wesley
"Inside Macintosh V", Apple Computer Inc., Addison Wesley

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

1. The current implementation is highly 8-bit dependent, but this doesn't make it hard to implement on 24 or 32 bit color systems, since these group pixel in 1-byte (8 bit) groups.
2. You can use "patXor" mode to invert selected bits, but this is the only transfer mode that leaves some bitplanes unaffected.
3. Normally VAInit calls SetPort(VA.window), so you are all set for QuickDraw drawing.
4. You can also use the built-in constants BG1 and BG2 as PmForeColor values. BGC is 255 and it erases all bitplanes.

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

Created: Saturday, September 29, 1990

Last change: Tuesday, March 12, 1991

Project STORM

Author: Juri Munkki

Vector Animation Toolkit

This document describes the programming interface to a real-time animation toolkit that allows vectors and various kind of points as the basic elements of animation.