I M P U L S E

Portable Graphics Engine
AlphaMask, Inc.
March 1, 2000

Overview

Impulse is a cross-platform graphics engine, capable of rendering high quality graphics on PCs, Macs, Unix, and Embedded systems. It is written entirely in C++, though its API can easily be wrapped in C functions for use by C-only clients.

Impulse runs well on popular systems (PC, Mac, Unix), but it has minimal knowledge of the host OS. Access to the file system (for disk-based Fonts) and simple memory allocation is all that is required. Font support is a separate module within Impulse, and can be replaced with custom modules, making it very easy to port to custom OS environments.

This document is intended as an introduction to Impulse. It describes the major classes available in the API, and how they work together. To help organize the classes and concepts, this document breaks the process of drawing into three elements.

- **Device**: This is where the drawing takes place, typically a bitmap (either the screen or an offscreen buffer), but it can be simply a redirection for recording the drawing into a stream, a postscript file, or a redirection for computing bounding boxes or performing hit-testing. The device also contains a view stack consisting of a matrices and clips.
- **Attribute**: This object is a collection of all the drawing attributes that affect the color and style of the drawing. It contains flags for antialiasing and filtering, and fields for frame size, text font and size, and special effects like gradients, blurs, extrudes.
- **Primitive**: This is the actual object being drawn. Impulse supports 3 classes of drawing primitives: Geometric (lines, rectangles, paths), Bitmap, and Text. Other common shapes (oval, round-rect, arc) can easily be constructed using paths.

GEOMETRY

Coordinates

Impulse is a device-independent graphics engine. This means that drawing primitives are specified in ideal or local space (fractional values), and may be transformed (scaled, rotated, etc) before they are used to draw something on the device (in device or pixel space). Impulse coordinates have X increasing from left to right, and Y increasing from top to bottom. The top-left corner of a device defaults to (0, 0).

Impulse can be conditionally compiled to use or not use floating point numbers. To facilitate this, all Impulse coordinates use the type **hsScalar**, denoting a 32bit fractional value that may be a float or a 16.16 fixed.

```
typedef Int32 hsFixed;  // 16.16
#define hsFixed1        (1 << 16)

#if HS_SCALAR_IS_FLOAT
    typedef float hsScalar;
    #define hsScalar1   float(1)
#else
    typedef hsFixed hsScalar;
    #define hsScalar    hsFixed1
#endif
```

Some operations with **hsScalars** can be performed directly in C, such as assignment, addition/subtraction, comparison, as well as some (but not all) operations with integers.

```
hsScalar a, b, c, d;

a = b + c - d;
if (a > b) c = d;

a = b * 3;    // legal
a = b / 3;    // legal
a = b + 3;    // ILLEGAL! Must use a = b + hsIntToScalar(3)
a = 3;        // ILLEGAL! Must use a = hsIntToScalar(3)
```

However, other operations require macros to insure that the operations work correctly with either version of **hsScalar**: e.g. multiplication/division, conversion to and from integers.

```
hsScalar a, b, c;
int      i;

a = hsIntToScalar(i);
a = hsScalarMul(b, c);    //  b * c
a = hsScalarDiv(b, c);    //  b / c
i = hsScalarRound(a);     //  rounds
i = hsScalarToInt(a);     //  truncates
```

These types and macros are found in hsFixedTypes.h and hsScalar.h

Points and Rectangles

Points consist of two values: X and Y. Impulse defines a scalar point as **hsPoint2**, and an integer point as **hsIntPoint2** (defined in hsPoint2.h).

```
struct hsPoint2 {
    hsScalar fX, fY;
};

struct hsIntPoint2 {
    Int32    fX, fY;
};
```

Rectangles consist of four values: Left, Top, Right, Bottom. Impulse defines a scalar rect as **hsRect**, and an integer rect as **hsIntRect** (in hsRect.h).

```
struct hsRect {
    hsScalar fLeft, fTop, fRight, fBottom;
};

struct hsIntRect {
    Int32    fLeft, fTop, fRight, fBottom;
};
```

For a rectangle to be valid, fLeft <= fRight and fTop <= fBottom.

Rects can be used as a drawing primitive, and they can also (along with Paths) be used as a clip.

Matrices

All geometric transformations (translate, scale, rotate, etc.) are specified with matrices. Impulse defines a 3x3 matrix using hsScalars.

```
struct hsMatrix33 {
    hsScalar fMap[3][3];
};
```

While the client may set the values for fMap directly, there are a host of methods designed to help you with this.

```
Reset();
SetTranslate(hsScalar dx, hsScalar dy);
SetScale(hsScalar scaleX, hsScalar scaleY,
         hsScalar pivotX, hsScalar pivotY);
SetRotate(hsScalar angle,
          hsScalar pivotX, hsScalar pivotY);
SetSkew(hsScalar skewX, hsScalar skewY,
        hsScalar pivotX, hsScalar pivotY);
```

These Set_ methods initialize the matrix to the state specified by the parameters.

NOTE: The pivot parameters specify what coordinate should be left unchanged by the matrix. For example, to rotate about the point P, use SetRotate(degrees, P.fX, P.fY).

```
Translate(hsScalar dx, hsScalar dy);
Scale(hsScalar scaleX, hsScalar scaleY,
      hsScalar pivotX, hsScalar pivotY);
Rotate(hsScalar angle,
       hsScalar pivotX, hsScalar pivotY);
Skew(hsScalar skewX, hsScalar skewY,
     hsScalar pivotX, hsScalar pivotY);
```

All of these methods modify the matrix by the specified parameters. Thus, they should not be called on an uninitialized matrix.

To concatenate two matrices together, use SetConcat. This produces a matrix that applies both transformations at once. NOTE: The order of the matrices is important. The resulting matrix effectly applies the second matrix and then the first.

```
SetConcat(const hsMatrix33* matrixA,
          const hsMatrix33* matrixB);

// For example

hsMatrix33  a, b, c;

a.SetScale(hsIntToScalar(3), hsIntToScalar(3), 0, 0);
b.SetTranslate(hsIntToScalar(10), hsIntToScalar(20));

c.SetConcat(&a, &b);   // c will translate and then scale
c.SetConcat(&b, &a);   // c will scale and then translate
```

Paths

Paths are opaque objects, used to store geometry more complex than just a rectangle. Paths can contain multiple contours, and each contour can be made up of any number of line and curve segments. The curve segments in a path are cubic beziers.

```
class hsPath;
```

A path is created by making method calls to add lines and curves.

```
void   MoveTo(const hsPoint2& pt);
void   LineTo(const hsPoint2& pt);
void   CurveTo(const hsPoint2& pt0, const hsPoint2& p1,
               const hsPoint2& p2);
void   Close();  // close the current contour
```

There are no methods for deleting segments within a path. However, you can clear the entire path using Reset().

```
void   Reset();
hsBool IsEmpty() const;
```

Paths can be drawn using either the even-odd (EO) rule, or the winding rule. This is specified in the path with the kEOFill_PathFlag. Paths default to winding fill (flags == 0).

```
enum {
    kEOFill_PathFlag = 0x01
};

UInt32 GetFlags() const;
void   SetFlags(UInt32 flags);
```

Paths have helper methods for adding common shapes as contours.

```
void  AddRect(const hsRect* rect);
void  AddPoly(int count, const hsPoint2 pts[]);
void  AddPath(const hsPath* path);
void  AddOval(const hsRect* oval);
void  AddCircle(hsScalar cX, hsScalar cY, hsScalar radius);
void  AddRRect(const hsRect* r, hsScalar w, hsScalar h);
void  AddArc(const hsRect* r, hsScalar startAngle,
             hsScalar sweepAngle, hsBool wedge);
```

Paths can return their bounds (as a rectangle), and be transformed by a matrix.

```
void  GetBounds(hsRect* bounds, hsBool exact) const;
void  Transform(const hsMatrix33* matrix);
void  Translate(hsScalar dx, hsScalar dy);
```

Paths can be used as a drawing primitive, and they can also (along with Rects) be used as a clip.

PathIterator

Since paths are opaque, Impulse provides an iterator for retrieving the data inside.

```
class hsPathIterator {
public:
                hsPathIterator(const hsPath* path,
                               hsBool forceClosed);

    hsPath::Verb Next(hsPoint2 pts[4]);
};
```

The Next() method is called in a loop, until it returns kDone_PathVerb. The interpretation of the pts[] parameter depends on the return value.

Verb returned from Next()	Pts[] assigned
kDone_PathVerb	none
kMoveTo_PathVerb	pts[0]

kLineTo_PathVerb	pts[0..1]
kCurveTo_PathVerb	pts[0..3]
kClose_PathVerb	none

Example:

```
hsPathIterator iter(&path);
hsPath::Verb   verb;
hsPoint2       pts[4];

while ((verb = iter.Next(pts)) != hsPath::kDone_PathVerb)
{
    switch (verb) {
    case hsPath::kMoveTo_PathVerb:
        // pts[0] begins a new contour
        break;
    case hsPath::kLineTo_PathVerb:
        // pts[0..1] are a line segment
        break;
    case hsPath::kCurveTo_PathVerb:
        // pts[0…3] are a bezier segment
        break;
    case hsPath::kClose_PathVerb:
        // marks the current contour closed
        break;
}
```

Bitmaps

While a bitmap isn't exactly geometry, it does represent the structure and dimensions of a drawing primitive, so it is discussed here.

```
class hsGBitmap {
public:
    enum Config {
        kNoConfig,
        kARGB32Config,
        kRGB32Config,
        k555Config,
        kIndex8Config,
        kAlpha8Config
    };
    enum {
        kOddFieldFlag  = 0x01,
        kEvenFieldFlag = 0x02
    };

                    hsGBitmap();
                    ~hsGBitmap();

    void*           fImage;
    UInt32          fWidth, fHeight, fRowBytes;
```

```
    Config          GetConfig() const;
    void            SetConfig(Config config);

    UInt32          GetFlags() const;
    void            SetFlags(UInt32 flags);

    hsGColorTable*  GetColorTable() const;
    void            SetColorTable(hsGColorTable* ctable);

    unsigned        GetPixelSize() const;
};
```

A **hsGBitmap** does not own the memory for the pixels, but merely points to it. It is the responsibility of the client to manage the pixel memory. The fields of a bitmap are:

- fImage: Points to the memory for the pixels.
- fWidth, fHeight: Dimensions of the bitmap.
- fRowBytes: The number of bytes between subsequent rows of pixels.

Bitmaps are oriented top-to-bottom. Thus the first pixel pointed to by fImage corresponds to the top-left corner of the bitmap.

The color-table class is a descendent of hsRefCnt, and is used with kIndex8Config to map 8-bit indices (the pixel values) to colors.

NOTE: Impulse treats 32-bit pixels with alpha as **premultiplied** colors. This means that within each pixel, the RGB components are stored already scaled by their alpha component. This applies to bitmaps that are drawn as primitives, as well as the result of Impulse drawing into a bitmap.

Color	**32-bit format [ARGB]**
Black	[0xFF 0 0 0]
White	[0xFF 0xFF 0xFF 0xFF]
Red	[0xFF 0xFF 0 0]
50% Translucent Red	[0x80 0x80 0 0]
Transparent	[0 0 0 0]

A rule of thumb for premultiplied colors: all color components must be <= the alpha component.

ATTRIBUTES

RefCnt

```
class hsGAttribute : public hsRefCnt;
```

The **hsGAttribute** object is derived from **hsRefCnt**. This allows the attribute object to be safely referenced by multiple objects.

```
class hsRefCnt {
    Int32        fRefCnt;
public:
                 hsRefCnt() : fRefCnt(1) {}
    virtual      ~hsRefcnt();

    virtual void Ref();
    virtual void UnRef();
};
```

When a **hsRefCnt** object is created, its private counter is initialized to 1. Each time Ref() is called, the counter is incremented. Each time UnRef() is called, the counter is decremented. If the counter gets to 0, then the object is deleted. It is an error to explicitly delete a hsRefCnt object whose counter is > 1.

NOTES:

For the following sections, enums and methods will be listed without the hsGAttribute:: prefix, but they are all defined inside the hsGAttribute class (see hsGAttribute.h).

All of the Set… methods return a boolean value indicating whether the method actually changed the setting. If the specified value is the same as the one already in the attribute, the method returns FALSE, else the setting is changed and the method returns TRUE.

Attribute Flags

Attribute flags specify various options for modifying a drawing. The default setting is a value of 0.

```
enum {
    kAntiAlias         = 0x01,
    kFrame             = 0x02,
    kFilterBitmap      = 0x04,
    kSquarePen         = 0x08,
    kKernText          = 0x10,
    kSubPixelText      = 0x20,
    kLinearMetricsText = 0x40,
    kLinearContourText = 0x80
};

UInt32 GetFlags() const;
hsBool SetFlags(UInt32 flags);
```

Changing the value of kFrame may be done quite often. To accommodate this, two helper methods are available.

```
hsBool SetFillMode();    // clear the kFrame bit
hsBool SetFrameMode();   // set the kFrame bit
```

Attribute Color

Color is specified in 16-bit component ARGB, represented by **hsGColor**. For alpha, 0 specifies transparent, and 0xFFFF specifies opaque. There is a single color in the attribute, and it applies to all primitives (line, rectangle, path, text) except for bitmaps, which only respect the color's alpha value.

```
typedef UInt16 hsGColorValue;

struct hsGColor {
    hsGColorValue fA, fR, fG, fB;
};

void   GetColor(hsGColor* color) const;
hsBool SetColor(const hsGColor* color);
hsBool SetARGB(hsGColorValue alpha, hsGColorValue red,
               hsGColorValue green, hsGColorValue blue);
```

Along with the color, two other objects can affect the color of the resulting image. **hsGShader** is a client-specified object that supplies per-pixel colors. It is called for each scanline of the primitive being drawn. **hsGXferMode** also is called per scanline, and is responsible for compositing the source colors onto the device. Each of these objects are optional, and may be nil.

```
class hsGShader : public hsRefCnt;
class hsGXferMode : public hsRefCnt;

hsGShader*   GetShader() const;
hsBool       SetShader(hsGShader* shader);

hsGXferMode* GetXferMode() const;
hsBool       SetXferMode(hsGXferMode* mode);
```

hsGShaders and hsGXferModes are derived from hsRefCnt, and are therefore reference counted. SetShader() and SetXferMode() automatically call Ref() on the new object (if it is not nil), and call UnRef() on the previous object (if it is not nil). GetShader() and GetXferMode() do not change the object’s reference count.

Example:

```
hsGShader* shader = new MyShader();

// shader’s refcnt is now 1
state->SetShader(shader);
// shader’s refcnt is now 2
```

```
shader->UnRef();
// shader's refcnt is now 1
(void)state->GetShader();
// shader's refcnt is still 1
state->SetShader(nil);
// shader is now deleted, since its refcnt went to 0
```

For special effects such as blurring or embossing, the client may provide a subclass of **hsGMaskFilter**. This object, when present, is called to modify the alpha mask of a drawing primitive. Like hsGShader and hsGXferMode, the hsGMaskFilter is reference counted.

```
class hsGMaskFilter : public hsRefCnt;

hsGMaskFilter* GetMaskFilter() const;
hsBool         SetMaskFilter(hsGMaskFilter* filter);
```

Attribute Framing

Geometric primitives can be draw filled or framed (stroked). If they are framed (kFrame bit is set), then the following fields apply.

```
enum CapType { kButtCap, kRoundCap, kSquareCap };
enum JoinType { kMiterJoin, kRoundJoin, kBluntJoin };

hsScalar GetFrameSize() const;
hsBool   SetFrameSize(hsScalar size);

CapTyp   GetCapType() const;
hsBool   SetCapType(CapType captype);

JoinType GetJoinType() const;
hsBool   SetJoinType(JoinType jointype);

hsScalar GetMiterLimit() const;
hsBool   SetMiterLImit(hsScalar limit);

hsScalar GetMinWidth() const;
hsBool   SetMinWidth(hsScalar minWidth);
```

The interpretation for FrameSize, CapType, JoinType and MiterLimit is the same as in PostScript. MinWidth allows the client to set the minimum size (in pixels) for a framed geometry. This is be used to keep very thin lines from disappearing when they are scaled down. If MinWidth is set to 0 (its default), no minimum thickness is enforced.

Clients may modify the geometry at draw time by providing a subclass of **hsGPathEffect**. This object is passed the original geometry, and may return a new one. Like hsGShaders, hsGXferModes, and hsGMaskFilters, this class is reference counted.

```
class hsGPathEffect : public hsRefCnt;

hsGPathEffect* GetPathEffect() const;
```

```
hsBool          SetPathEffect(hsGPathEffect* effect);
```

Clients may also override the scan conversion process by providing a subclass of **hsGRasterizer**. This object is passed a path, and returns an alpha mask. This object is reference counted like hsGShaders, hsGXferModes, hsGMaskFilters, and hsGPathEffects.

```
class hsGRasterizer : public hsRefCnt;

hsGRasterizer* GetRasterizer() const;
hsBool         SetRasterizer(hsGRasterizer* raster);
```

Attribute Text

Attributes for text include font, size, encoding, algorithmic styles, and spacing.

```
enum TextEncoding {
    kAsciiEncoding,
    kUTF8Encoding,
    kUnicodeEncoding
};

TextEncoding GetTextEncoding() const;
hsBool       SetTextEncoding(TextEncoding encoding);
```

The text encoding identifies what kind of character codes are passed to drawing and measuring methods. ASCII specifies that all character codes are 1-byte. UTF8 specifies that the characters require a variable number of bytes. Unicode specifies that each character is 16-bits.

```
typedef UInt32 hsGFontID;

hsGFontID    GetFontID() const;
hsBool       SetFontID(hsGFontID fontID);
```

Fonts are identified by a 32-bit font ID. These IDs are obtained using the hsGFontList methods. A value of 0 specifies that the default font should be used.

```
hsScalar     GetTextSize() const;
hsBool       SetTextSize(hsScalar textSize);
```

The text size specifies the size of the text (to be modified by the matrix and optional TextFace). Note that the size is an hsScalar, and may be a fractional value (e.g. 12.75).

These next two attributes (textface and textspacing) are optional structs. The Get methods return a boolean indicating if the attribute has the value. To clear the value, pass nil to the Set method.

```
struct hsGTextSpacing {
    enum {
        // trim spaces when justified
        kTrimJustText = 0x01
```

```
    };
    UInt32   fFlags;
    hsScalar fAlignment;   // < 0 means full justified
    hsScalar fSpaceExtra;  // ignore if fAlignment < 0
    hsScalar fCharExtra;   // ignore if fAlignment < 0
};

hsBool GetTextSpacing(hsGTextSpacing* spacing) const;
hsBool SetTextSpacing(const hsGTextSpacing* spacing);
```

hsGTextSpacing allows the client to override the character spacing and alignment when drawn using DrawGlyphs. fAlignment specifies a continuum between left (0), center (0.5) and right (1) alignment. If alignment is < 0, then its absolute value is interpreted as a width, and the text spacing is automatically adjusted to fit the text within that width. If alignment >= 0, then fSpaceExtra and fCharExtra are added to their respective characters. If the hsGTextSpacing field is nil (the default), text is drawn left-aligned.

The default setting for an attribute is no hsGTextSpacing. In this case, GetTextSpacing() returns false, and does not modify the face parameter. To reset the attribute to its default state, pass nil to SetTextSpacing().

```
struct hsGTextFace {
    hsScalar fBoldness;           // default hsScalar1
    hsScalar fSkew;               // default 0
    hsScalar fXScale;             // default hsScalar1
    hsScalar fXOffset;            // default 0
    hsScalar fOutlineWidth;       // default 0
    hsScalar fUnderlineThickness; // default 0
    hsScalar fUnderlineOffset;    // default 0
};

hsBool GetTextFace(hsGTextFace* face) const;
hsBool SetTextFace(const hsGTextFace* face);
```

hsGTextFace allows the client to modify the size and shape of the text. fBoldness specifies algorithmic emboldening. fSkew and fXScale combine to create a matrix that modifies the shape of the text. fXOffset adds itself to each character's advance width. fOutlineWidth specifies the thickness of outline text (a value of 0 means normal text). Underline thickness and offset specify where to draw an underline.

The default setting for attribute is no hsGTextFace. In this case, GetTextFace() returns false, and does not modify the face parameter. To reset the attribute to its default state, pass nil to SetTextFace().

Attribute Text Measure

MeasureGlyphs returns the width of a string, and returns the line height in two optional parameters. The character codes in the text parameter are interpreted based on the current TextEncoding.

```
hsScalar MeasureText(UInt32 length, const void* text,
```

```
                    hsPoint2* ascent, hsPoint2* descent);
```

Ascent and descent are points, so that MeasureText can return information about the angle of the text as well. The Y component of ascent and descent indicates the line height (above and below the baseline), and the X component reflects the italic angle (if any). For normal upright text, the X component is 0.

GetTextWidths returns an array of widths for each character in a string. The method returns the number of characters processed, base on the current TextEncoding. For kAsciiEncoding, the return value == length. For kUnicodeEncoding, the return value == length/2. For kUTF8Encoding, the value depends on the actual characters in the text.

```
int   GetTextWidths(UInt32 length, const void* text,
                    hsScalar widths[]);
```

GetTextPath converts the text into a path containing the outlines of all the characters.

```
void  GetTextPath(UInt32 length, const void* text,
                  hsPath* path);
```

GetTextPath returns the path scaled by the text-size (and any TextFace scaling), and filters it through the attributes PathEffect (if any).

DEVICE

The base class for all drawing devices is hsGDevice.

```
class hsGDevice : public hsRefCnt {
public:
    virtual void  Save();
    virtual void  Restore();

    virtual void  Concat(const hsMatrix33* matrix);
    virtual void  ClipPath(const hsPath* path);
    virtual hsMatrix33* GetTotalMatrix(hsMatrix33* matrix);
    virtual void  PushInto(hsGDevice* target) const;

    // The draw methods do nothing, but rely on
    // subclasses to provide the functionality

    virtual void  DrawFull(hsGAttribute* attr);
    virtual void  DrawLine(const hsPoint2* start,
                           const hsPoint2* stop,
                           hsGAttribute* attr);
    virtual void  DrawRect(const hsRect* rect,
                           hsGAttribute* attr);
    virtual void  DrawPath(const hsPath* path,
                           hsGAttribute* attr);
    virtual void  DrawBitmap(const hsGBitmap* b,
                           hsScalar x, hsScalar y,
                           hsGAttribute* attr);
    virtual void  DrawParamText(UInt32 length,
                           const void* text,
                           hsScalar x, hsScalar y,
                           hsGAttribute* attr);
    virtual void  DrawPosText(UInt32 length,
                           const void* text,
                           const hsPoint2 pos[],
                           const hsPoint2 tan[],
                           hsGAttribute* attr);
};
```

Device View Stack

The device maintains an internal stack of matrices and clips (views). These affect all primitives drawn into the device. A new "view" is pushed onto the stack when Save() is called. It is initialized to an identity matrix and an unrestricted clip. This new view can be modified: the matrix is changed using Concat(), and the clip is augmented by using ClipPath(). To pop the current view off the stack, call Restore().

Example:

```
device->DrawRect(&rect, &attr);
device->Save();
// now there is another view on the stack
device->Rotate(hsIntToScalar(30), 0, 0);
device->DrawRect(&rect, &attr);
// now the rect is rotated 30 degress about (0,0)
path.AddOval(&rect);
device->ClipPath(&path);
device->DrawRect(&rect, &attr);
// now the rect draws through an oval clip
device->Restore();
// now the device is back to its original view state
```

There are helper methods for manipulating the matrix and clip.

```
// balance with one call to Restore()

void  ClipRect(const hsRect* rect);

void  Translate(hsScalar dx, hsScalar dy);
void  Scale(hsScalar sx, hsScalar sy,
            hsScalar px, hsScalar py);
void  Rotate(hsScalar degrees, hsScalar px, hsScalar py);
void  Skew(hsScalar sx, hsScalar sy,
           hsScalar px, hsScalar py);
```

PushInto() is used to transfer the entire view stack from the source device. This is useful when you want to replicate the drawing from one device into another. Internally, this is done by first calling Save(), and then concatenating all of the matrices and clips from source. To restore the device to its state before the PushInto() call, only one call to Restore() is needed.

ClipRect() is a utility method for creating a rectangular path, and clipping with it. Internally, the code looks something like the following:

```
void hsGDevice::ClipRect(const hsRect* rect)
{
    if (rect != nil)
    {
        hsPath path;

        path.AddRect(rect);
        this->ClipPath(&path);
    }
}
```

Internally, Impulse detects paths that are rectangular, and uses them as such for efficiency.

Translate(), Scale(), Rotate(), Skew() and Concat() methods should look familiar. They are similar to the methods on hsMatrix33, except that on a Device, they premultiply the device matrix (are applied before the rest of the Device matrix), where as the hsMatrix33 methods postmultiply, applying their change after the original matrix.

Device TotalMatrix

The device method GetMatrix() returns only the current matrix for the view on the top of the stack. This is the matrix you are allow to modify. However, when a primitive is drawn, it is transformed by the concatenation of all of the matrices in the stack. This concatenated matrix is called the TotalMatrix. The TotalMatrix cannot be modified, but may be retrieved. It is useful for mapping (transforming) points into device space (pixel space in the case of a hsGRasterDevice).

```
hsMatrix33* GetTotalMatrix(hsMatrix33* matrix);
void        MapPoints(int count, const hsPoint2 src[],
                      hsPoint2 dst[]);
void        MapRect(const hsRect* src, hsRect* dst);
```

GetTotalMatrix() returns the parameter it is passed, not the actual total matrix. This allows the following usage.

```
hsMatrix33 matrix;

device->GetTotalMatrix(&matrix)->MapPoints(4, src, dst);
```

MapPoints() can accept *src*[] and *dst*[] being the same array. MapRect returns in *dst* the bounds of the transformed *src* rectangle in the case the TotalMatrix involves more than just translation and scaling.

Sometimes it is helpful to perform the inverse operation: mapping points (and vectors) from device coordinates back through the TotalMatrix. This can be done by calling GetTotalMatrix() and then inverting the matrix, or using the following helper methods:

```
hsBool  GetTotalInverse(hsMatrix33* inverse);
hsBool  InvertPoints(int count, const hsPoint2 src[],
                     hsPoint2 dst[]);
hsBool  InvertRect(const hsRect* src, hsRect* dst);
```

These inverse methods return a boolean value, indicating their success or failure. If the device's total matrix is non-invertible, these methods return false and do not modify their parameters.

Device Subclasses

Impulse provides several basic subclass of hsGDevice, overriding the above Draw_ methods.

- **hsGRasterDevice**: This subclass renders into a bitmap. The client can provide the memory for the bitmap, or the class can allocate it.
  - **hsGOffscreenDevice**: This subclass of hsGRasterDevice manages creating a platform-specific offscreen bitmap, and offers easy methods for copying it onto the screen.
- **hsGStreamDevice**: This subclass captures the drawing commands and writes them into a stream for later playback.

- **hsGPostScriptDevice**: This subclass captures the drawing commands and translates them into PostScript commands, ignoring those features of Impulse that are not supported in PostScript.
- **hsGHitTestDevice**: This subclass provides a device that tests whether a given point or rectangle intersects any of the primitives drawn into it.

Impulse also provides helper classes based around hsGDevice

- **hsGBounder**: This class provides a device that returns the bounds of any primitives drawn into it.
- **hsGStreamPlayback**: This class takes the drawing commands previously recorded by hsGStreamDevice into a stream, and replays them into another device.

hsGRasterDevice

To draw into a bitmap, use hsGRasterDevice (or its descendant hsGOffscreenDevice).

```
class hsGRasterDevice : public hsGDevice {
public:
    HSScanHandler*  GetHandler() const;
    virtual void    SetHandler(HSScanHandler* handler);

    void            GetOrigin(hsIntPoint2* origin);
    virtual void    SetOrigin(int x, int y);

    hsGBitmap*      GetPixels(hsGBitmap* pixels) const;
    virtual void    SetPixels(const hsGBitmap* pixels);

    hsIntRect*      GetBounds(hsIntRect* bounds) const;
    void            SetBounds(const hsIntRect* bounds,
                              unsigned bitDepth);

    virtual void    Erase(const hsGColor* color);

    // overrides of the draw methods
};
```

HSScanHandler is an optional object that the raster device can reference. If the device references one, it is called with the device-space (transformed into device coordinates) primitive before it is drawn. If the handler returns TRUE, then drawing continues. If the handler returns FALSE, then nothing is drawn. This can be used to accumulate the bounds of objects being drawn, or to hide a cursor.

SetOrigin() affects the device's total matrix by apply a translate after all other transforms have been applied.

Call SetPixels() to give the device the bitmap it should draw into. If the fImage field of the bitmap is set to nil, then the device will allocate the memory for the bitmap (based on its width, height, pixel-size). If this is done, then the device will manage deleting that memory

when either the device is destroyed, or another to call to SetPixels() is made. Calling GetPixels() returns a bitmap whose fImage field reflects either the memory specified at the SetPixels() call, or the memory allocated by the device. It also calls SetOrigin() with the top-left of the bounds.

SetBounds() is a helper method. It takes a bounding rectangle and constructs a bitmap based on it and the specified bitDepth. In turn, it calls SetPixels() with a bitmap whose fImage field is nil, forcing the device to allocate the memory.

Erase() fills the device's bitmap with the specified color (including alpha). This method does not call any of the virtual Draw methods, but writes to the pixels directly, ignoring the matrix or clip.

Raster Drawing

The methods DrawLine(), DrawRect(), and DrawPath() operate in the following manner.

1. Prepare the geometry for scan conversion
   1.1. Apply the hsGPathEffect (if any) from the attribute.
   1.2. Stroke the geometry (if kFrame is specified by the attribute).
   1.3. Apply the total-matrix to the geometry, transforming it into device space.
2. Scan convert the geometry into an alpha mask, clipped to the bounds of the stack of device clips.
   2.1. Use the hsGRasterizer (if any) from the attribute, going from a geometry to a mask.
   2.2. Apply the hsGMaskFilter (if any) from the attribute, generating another mask.
3. Blit the mask into the pixels using the color from the attribute, clipped to the stack of device clips.
   3.1. Use the hsGShader (if any) from the attribute to obtain the colors (modified by the attribute's color's alpha).
   3.2. Use the hsGXferMode (if any) from the attribute to blend the colors with the device's pixels.

DrawBitmap() draws the bitmap primitive with its top-left corner specified by the X and Y parameters. The bitmap respects the specified matrix and clip, and the attribute’s color's alpha, and optional hsGXferMode. If the device's matrix causes the bitmap to be scaled, rotated, or otherwise transformed when it is drawn, then Impulse looks at the kFilterBitmap flag in the attribute. Filtering generally generates better results, but runs slower.

DrawParamText() and DrawPosText() offer two different ways to specify where to draw text. DrawParamText() just specifies the starting location, and relies on the spacing information in the font (and the optional hsGTextFace and hsGTextSpace fields of the attribute) to determine where to draw the characters. DrawPosText() specifies the position of each character (and optionally a tangent for each character). Both methods use the font and text size from the attribute.

hsGOffscreenDevice

```
class hsGOffscreenDevice : public hsGRasterDevice {
public:
    hsOffscreen fOffscreen;

    void  SetSize(int width, int height, int depth);
    void  CopyToScreen(int dx, int dy);
 };
```

This subclass of hsGRasterDevice creates an offscreen object to use as the pixels. How this is done depends on the host OS.

- Windows: the offscreen creates a HDC, and the CopyToScreen() method calls StretchDIBits() or SetDIBitsToDevice().
- Macintosh: the offscreen creates a Gworld, and the CopyToScreen() method uses CopyBits().

hsGStreamDevice

```
class hsGStreamDevice : public hsGDevice {
public:
    void  StartRecording(hsStream* outStream);
    void  StopRecording();

    // Overrides of the draw methods from hsGDevice
};
```

This device does not render anything, but instead records all of the drawing, matrix and clip calls into the stream object the caller provides (see hsStream.h). The resulting stream is completely self-contained, and can be copied or written to disk. To replay the drawing, simply pass the stream to a hsGStreamPlayback object.

```
class hsGStreamPlayback {
public:
          hsGStreamPlayback(hsRegistry* registry);

    void  Playback(hsStream* inStream, hsGDevice* target);
};
```

The optional hsRegistry object passed to the construct allows any flattened subclasses to be reanimated during playback. The file hsGRegisterAll.h declares a function that registers all of the features provided with Impulse (gradient shaders, dashing path-effects, etc.).

Example:

```
void DrawStream(hsStream* inStream, hsGDevice* target)
{
    hsRegistry         registry;
    hsGStreamPlayback  player(&registry);
```

```
        hsGRegisterAll(&registry);

        playback.Playback(inStream, target);
    }
```

The target device can be any subclass of hsGDevice, including another stream device. You may pass nil for the constructor of the hsGStreamPlayback, in which case any subclassed objects embedded in the stream will be ignored.

hsGPostScriptDevice

```
class hsGPostScriptDevice : public hsGDevice {
public:
    void SetPaperSize(int width, int height);
    void SetPageBounds(const hsIntRect* margins);

    void StartDoc(FILE* target);
    void StartPage();
    void EndPage();
    void EndDoc();

    // Overrides of the Draw methods from hsGDevice
};
```

The hsGPostScriptDevice, like hsGStreamDevice, captures all drawing commands into (in this case) a file. However, this device converts these commands into their PostScript equivalents. It ignores those Impulse features that have no corresponding feature in PostScript: hsGRasterizer, hsGMaskFilter, hsGShader, hsGXferMode. In addition, it does not offer any font-downloading services. It is up to the client to insure that any fonts needed will be available on the printer.

NOTE: This is an experimental class, and not all features are fully implemented.

hsGHitTestDevice

The hsGHitTestDevice class provides for pixel-accurate hit testing. It does this by storing a target rectangle in device (pixel) coordinates. Any drawing performed on the device will not render, but will set a flag as to whether its pixels intersected the target rectangle.

```
    class hsGHitTestDevice {
    public:
                hsGHitTestDevice(const hsIntRect* target,
                                 hsBool respectAlpha);
        virtual  ~hsGHitTestDevice();

        void     Reset();
        hsBool   IsHit() const;
    };
```

The target rectangle (specified by the client) is expressed in device coordinates. To specify a single point at (X,Y), pass the rectangle (X, Y, X+1, Y+1). Any drawing directed to this device will not render, but will be tested against the target rectangle. Once one primitive intersects the target rectangle, the IsHit() method will return true. Calling Reset() sets the IsHit() flag back to FALSE.

hsGDevice Utilities

The following classes are not subclasses of hsGDevice, but do create devices internally and offer functionality based on hsGDevice.

hsGBounder

The hsGBounder class provides a mechanism for calculating the bounds of one or drawing primitive. Note that this bounds can vary greatly from just the bounds of the primitive's geometry, for there are many factors that affect the bounds…

- Framing (stroking) adds to the bounds. In the simplest case, 1/2 of the frame size is added to each side of the bounds, but miter joins (if they are selected in the attribute) can extend the bounds even further.
- The device's matrix can transform the geometry, affecting its bounds.
- The optional objects hsGPathEffect, hsGRasterizer, hsGMaskFilter can all modify the drawing of a primitive such that its bounds differ from the geometry. Note that hsGShader and hsGXferMode objects cannot affect the size of the drawn primitive, only what color(s) it is drawn in.

```
class hsGBounder {
public:
    hsGDevice*  GetDevice();

    void        Reset();
    hsBool      GetBounds(hsIntRect* bounds);
};
```

GetDevice() returns a private device object. Any drawing directed to this device will not appear anywhere, but will its bounds will be accumulated by the bounder object. Calling Reset() reinitializes the bounder's accumulater rectangle, so the same bounder object can be used to compute the bounds of different primitives. Notice that GetBounds() returns a boolean, and returns the resulting bounds (if bounds != nil) as an integer rectangle. This is the device coordinate bounds of the primtive(s) that were drawn into the device returned by GetDevice(). If GetBounds() returns false, then no primitive was drawn into the device (or if one was, it was clipped out).

Example usage:

```
class Shape {
public:
    virtual void Draw(hsGDevice* device) = 0;
```

```
    hsBool       Bounds(hsIntRect* bounds);
};

hsBool Shape::Bounds(hsIntRect* bounds)
{
    hsGBounder bounder;

    this->Draw(bounder.GetDevice());

    return bounder.GetBounds(bounds);
}
```

This example assumes the the Shape class has subclasses that define the Draw() method for various types of shapes. Each shape subclass knows how to draw itself into a device. The Bounds() method is not virtual, and need only be implemented by the base class, since it can create a bounder device and pass that to the Shape's virtual Draw() method. Whatever the subclass draws will get accumulated by the bounder's device, and returned when GetBounds() is called.

We can add hit-testing to our example.

```
class Shape {
public:
    …
    hsBool       HitTest(int x, int y);
};

hsBool Shape::HitTest(int x, int y)
{
    hsIntRect    target;

    target.Set(x, y, x + 1, y + 1);

    hsGHitTestDevice tester(&target, true);

    this->Draw(&tester);

    return tester.IsHit();
}
```