Implementing Stereoscopic 3D in Your Applications

Room C | 09/20/2010 - 16:00 - 17:20

Samuel Gateau, NVIDIA, Steve Nash, NVIDIA
Agenda

- How It Works
- NVIDIA 3D Vision
- Implementation Example
- Stereoscopic Basics
- Depth Perception
- Parallax Budget
- Rendering in Stereo
- What’s Next?
3D vs Stereo

- "In 3D" is the new "Stereo"
  - They are used interchangeably, stereoscopic rendering is the technical means to make an image "In 3D"

- Each eye gets its own view rendered with a slightly different camera location: usually about as far apart as your eyes

- Stereo displays and APIs are used to manage these two views
Applications render a Left Eye view and Right Eye view with slight offset between.

A Stereoscopic display then shows the left eye view for even frames (0, 2, 4, etc) and the right eye view for odd frames (1, 3, 5, etc).
Stereoscopic Basics

How It Works

In this example active shutter glasses black-out the right lens when the left eye view is shown on the display and black-out the left lens when the right eye view is shown on the display.

This means that the refresh rate of the display is effectively cut in half for each eye. (e.g. a display running at 120 Hz is 60 Hz per eye)

The resulting image for the end user is a combined image that appears to have depth in front of and behind the stereoscopic 3D Display.
NVIDIA 3D Vision

Software
3D Vision SW automatically converts mono games to Stereo
Direct X only

Hardware
IR communication
3D Vision certified displays
Support for single screen or 1x3 configurations
## NVIDIA 3D Vision Pro

### Hardware
- RF communication
- 3D Vision certified displays, Passive Displays, CRTs and projectors
- Up to 8 displays
- Mix Stereo and Regular Displays
- G-Sync support for multiple displays and systems
- Direct connection to GPU mini-DIN

### Software
- Supports Consumer 3D Vision SW or Quad Buffered Stereo
- QBS: OpenGL or DirectX
  For DX QBS, e-mail 3DVisionPro_apps@nvidia.com for help
NVIDIA 3D Vision Pro

Hardware - cont’d

- Designed for multi-user professional installations
- No line of sight requirement, no dead spots, no cross talk
- RF bi-directional communication with UI
- 50m range
- Easily deploy in office no matter what the floor plan
Implementation Example
Implementation Example: OpenGL

Step 1: Configure for Stereo
Implementation Example: OpenGL
Step 2: Query and request PFD_STEREO

```c
iPixelFormat = DescribePixelFormat(hdc, 1,
sizeof(PIXELFORMATDESCRIPTOR), &pfd);
while (iPixelFormat) {

    DescribePixelFormat(hdc, iPixelFormat,
sizeof(PIXELFORMATDESCRIPTOR), &pfd);

    if (pfd.dwFlags & PFD_STEREO){
        iStereoPixelFormats++;
    }
    iPixelFormat--;
}
if (iStereoPixelFormats == 0)
    // no stereo pixel formats available
    StereoIsAvailable = FALSE;
else
    StereoIsAvailable = TRUE;
```
Implementation Example: OpenGL

Step 2 cont’d

if (StereoIsAvailable){
    ZeroMemory(&pfd, sizeof(PIXELFORMATDESCRIPTOR));
    pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
    pfd.nVersion = 1;
    pfd.dwFlags = PFD_DRAW_TO_WINDOW |
                   PFD_SUPPORT_OPENGL |
                   PFD_DOUBLEBUFFER |
                   PFD_STEREO;
    pfd.iPixelFormat = PFD_TYPE_RGBA;
    pfd.cColorBits = 24;

    iPixelFormat = ChoosePixelFormat(hdc, &pfd);

    if (iPixelFormat != 0){
        if (SetPixelFormat(hdc, iPixelFormat, &pfd)){
            hglrc = wglCreateContext(hdc);
            if (hglrc != NULL){
                if (wglMakeCurrent(hdc, hglrc)){
                    ...
    ...}
// Select back left buffer
glDrawBuffer(GL_BACK_LEFT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Setup the frustum for the left eye
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(Xmin - FrustumAssymmetry, 
    Xmax - FrustumAssymmetry, 
    -0.75, 0.75, 0.65, 4.0);

glTranslatef(eyeOffset, 0.0f, 0.0f);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

<Rendering calls>
// Select back right buffer
glDrawBuffer(GL_BACK_RIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Setup the frustum for the right eye.
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(Xmin + FrustumAssymmetry,
        Xmax + FrustumAssymmetry,
              -0.75, 0.75, 0.65, 4.0);
glTranslatef(-eyeOffset, 0.0f, 0.0f);
glTranslatef(0.0f, 0.0f, -PULL_BACK);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

<Rendering calls>
// Swaps both left and right buffers
SwapBuffers(hdc);
Changes to the rendering pipe

FROM MONO TO STEREO
In Mono

Scene is viewed from one eye and projected with a perspective projection along eye direction on Near plane in Viewport.
In Stereo

Scene

Near plane

Mono Eye

Eye space

X Y Z

In Stereo
In Stereo:

**Two eyes**

Left and Right eyes

Shifting the mono eye along the X axis
In Stereo:

**Two eyes**

Left and Right eyes
Shifting the mono eye along the X axis
Eye directions are parallels
In Stereo: Two Eyes, One Screen

Left and Right eyes
Shifting the mono eye along the X axis
Eye directions are parallels

One “virtual” screen
In Stereo: Two Eyes, One Screen

Left and Right eyes
Shifting the mono eye along the X axis
Eye directions are parallels

One “virtual” screen
Where the left and right frustums converge

Scene

Left Frustum

Right Frustum

Virtual Screen

Near plane

Left Eye
Mono Eye
Right Eye
In Stereo: Two Eyes, One Screen, Two Images

Left and Right eyes
Shifting the mono eye along the X axis
Eye directions are parallels

One “virtual” screen
Where the left and right frustums converge

Two images
2 images are generated at the near plane in each views
In Stereo: Two Eyes, One Screen, Two Images

Two images
2 images are generated at the near plane in each view
Presented independently to each eye of the user on the real screen
Stereoscopic Rendering

Render geometry twice
From left and right eyes
Into left and right images
Basic definitions so we all speak English

DEFINING STEREO PROJECTION
Stereo Projection

- Stereo projection matrix is a horizontally offset version of regular mono projection matrix
  - Offset Left / Right eyes along X axis
Stereo Projection

- Projection Direction is parallel to mono direction (NOT toed in)
- Left and Right frustums converge at virtual screen
**Interaxial**

- Distance between the 2 virtual eyes in eye space
- The mono, left & right eyes directions are all parallels
Separation

- The normalized version of \textit{interaxial} by the virtual screen width
- More details in a few slides....

Separation = \frac{\text{Interaxial}}{\text{Screen Width}}
Convergence

- Virtual Screen‘s depth in eye space ("Screen Depth")
- Plane where Left and Right Frustums intersect
Parallax

- Signed Distance on the virtual screen between the projected positions of one vertex in left and right image
- Parallax is function of the depth of the vertex
Where the magic happens and more equations

DEPTH PERCEPTION
Virtual vs. Real Screen

Parallax creates the depth perception for the user looking at the real screen presenting left and right images.

The virtual screen is perceived as the real screen.
In / Out of the Screen

<table>
<thead>
<tr>
<th>Vertex Depth</th>
<th>Parallax</th>
<th>Vertex Appears</th>
</tr>
</thead>
<tbody>
<tr>
<td>Further than Convergence</td>
<td>Positive</td>
<td>In the Screen</td>
</tr>
<tr>
<td>Equal Convergence</td>
<td>Zero</td>
<td>At the Screen</td>
</tr>
<tr>
<td>Closer than Convergence</td>
<td>Negative</td>
<td>Out of the Screen</td>
</tr>
</tbody>
</table>

**Diagram:**
- **Vertex Depth:**
  - Further than Convergence: Positive, In the Screen
  - Equal Convergence: Zero, At the Screen
  - Closer than Convergence: Negative, Out of the Screen

- **Parallax:**
  - Positive: In the Screen
  - Zero: At the Screen
  - Negative: Out of the Screen

- **Convergence:**
  - Out of the Screen
  - Screen
  - In the Screen

**Eye space:**
- Left Eye
- Mono Eye
- Right Eye

**Diagram notes:**
- **Eye space:** X, Y, Z
- **Convergence**
Parallax in normalized image space

Parallax = Separation * ( 1 – Convergence / W )

- Parallax diverges quickly to negative infinity for object closer to the eye
- Parallax is 0 at screen depth
- Maximum Parallax at infinity is the separation (distance between the eyes)
Eye Separation

- **Interocular** (distance between the eyes) is on average 2.5” ⇐ 6.5 cm
- Equivalent to the visible parallax on screen for objects at infinity
- Depending on the screen width, we define a normalized “Eye Separation”

\[
\text{Eye Separation} = \frac{\text{Interocular}}{\text{Real Screen Width}}
\]

- Different for each screen model
- A reference maximum value for the **Separation** used in the stereo projection for a comfortable experience
Separation should be Comfortable

- The maximum parallax at infinity is Separation
- Eye Separation is an average, should be used as the very maximum Separation value
  - Never make the viewer look diverge
  - People don’t have the same eyes
- For Interactive application, let the user adjust Separation
  - When the screen is close to the user (PC scenario) most of the users cannot handle more than 50% of the Eye Separation
Eye Separation is the Maximum Comfort Separation
Safe Parallax Range

Parallax vs Depth

- Eye Separation
- Separation 1
- Separation 2
- Convergence

NVIDIA
PARALLAX BUDGET
Parallax Budget
How much parallax variation is used in the frame

Parallax budget
Separation
Nearest pixel
Farthest pixel
Convergence
In Screen: Farthest Pixel

- At 100 * Convergence, Parallax is 99% of the Separation
  - For pixels further than 100 * Convergence, Elements looks flat on the far distance with no depth differentiation

- Between 10 to 100 * Convergence, Parallax vary of only 9%
  - Objects in that range have a subtle depth differentiation
Out of the Screen: Nearest pixel

- At Convergence / 2, Parallax is equal to -Separation, out of the screen
  - Parallax is very large (> Separation) and can cause eye strains
Convergence sets the scene in the screen
Defines the window into the virtual space
Defines the style of stereo effect achieved (in / out of the screen)
Separation scales the parallax budget

Scales the depth perception of the frame
Adjust Convergence

- Convergence must be controlled by the application
- Camera parameter driven by the look of the frame
  - Artistic / Gameplay decision
  - Should adjust for each camera shot / mode
    - Make sure the scene elements are in the range \([\text{Convergence} / 2, 100 * \text{Convergence}]\)
  - Adjust it to use the Parallax Budget properly
    - Cf Bob Whitehill Talk (Pixar Stereographer) at Siggraph 2010
- Dynamic Convergence is a bad idea
  - Except for specific transition cases
  - Analyze frame depth through an Histogram and focus points?
    - Ongoing projects at NV
Let’s do it

RENDERING IN STEREO
## Stereoscopic Rendering

<table>
<thead>
<tr>
<th>Render geometry <strong>twice</strong></th>
<th>Do <strong>stereo drawcalls</strong></th>
<th>Duplicate <strong>drawcalls</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>From left and right <strong>eyes</strong></td>
<td>Apply <strong>stereo projection</strong></td>
<td>Modify <strong>projection matrix</strong></td>
</tr>
<tr>
<td>Into left and right <strong>images</strong></td>
<td>Use <strong>stereo surfaces</strong></td>
<td>Duplicate <strong>render surfaces</strong></td>
</tr>
</tbody>
</table>
How to implement stereo projection?

- Fully defined by mono projection and Separation & Convergence
- Replace the perspective projection matrix by an offset perspective projection
  - horizontal offset of Interaxial
  - Negative for Right eye
  - Positive for Left eye
- Or just before rasterization in the vertex shader, offset the clip position by the parallax amount (Nvidia 3D vision driver solution)

\[ \text{clipPos.x} += \text{EyeSign} \times \text{Separation} \times (\text{clipPos.w} - \text{Convergence}) \]

EyeSign = +1 for right, -1 for left
Stereo Transformation Pipeline

**Standard Mono**

1. **World space** → **View Transform** → **Eye space** → **Projection Transform** → **Clip space** → **Perspective Divide** → **Normalized space** → **Viewport Transform** → **Image space**

**Stereo Projection Matrix**

1. **Eye Space** → **Stereo Projection Transform** → **Stereo Clip space** → **Perspective Divide** → **Stereo Normalized space** → **Viewport Transform** → **Stereo Image space**

**Stereo Separation on clip position**

1. **Eye space** → **Projection Transform** → **Clip space** → **Stereo Separation** → **Stereo Clip space** → **Perspective Divide** → **Stereo Normalized space** → **Viewport Transform** → **Stereo Image space**
Stereo rendering surfaces

- View dependent render targets must be duplicated
  - Back buffer
  - Depth Stencil buffer
- Intermediate full screen render targets used to process final image
  - High dynamic range, Blur, Bloom
  - Screen Space Ambient Occlusion
Mono rendering surfaces

- View independent render targets DON’T need to be duplicated
  - Shadow map
  - Spot light maps projected in the scene
How to do the stereo drawcalls?

- Simply draw the geometries twice, in left and right versions of stereo surfaces
- Can be executed per scene pass
  - Draw left frame completely
  - Then Draw right frame completely
  - Need to modify the rendering loop
- Or for each individual objects
  - Bind Left Render target, Setup state for left projection, Draw geometry
  - Bind Right render target, Setup state for right projection, Draw Geometry
  - Might be less intrusive in an engine
- Not everything in the scene needs to be drawn
  - Just depends on the render target type
# When to do what?

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Render Target Type</th>
<th>Stereo Projection</th>
<th>Stereo Drawcalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shadow maps</td>
<td>Mono</td>
<td>No</td>
<td>Draw Once</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use Shadow projection</td>
<td></td>
</tr>
<tr>
<td>Main frame</td>
<td>Stereo</td>
<td>Yes</td>
<td>Draw Twice</td>
</tr>
<tr>
<td>Any Forward rendering pass</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reflection maps</td>
<td>Stereo</td>
<td>Yes</td>
<td>Draw Twice</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Generate a stereo</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reflection projection</td>
<td></td>
</tr>
<tr>
<td>Post processing effect</td>
<td>Stereo</td>
<td>No</td>
<td>Draw Twice</td>
</tr>
<tr>
<td>(Drawing a full screen quad)</td>
<td></td>
<td>No Projection needed at all</td>
<td></td>
</tr>
<tr>
<td>Deferred shading lighting pass</td>
<td>Stereo G-buffers</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>(Drawing a full screen quad)</td>
<td></td>
<td>Be careful of the Unprojection</td>
<td>Should be stereo</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What could go possibly wrong?

EVERYTHING IS UNDER CONTROL
3D Objects

- All the 3D objects in the scene should be rendered using a unique Perspective Projection in a given frame
- All the 3D objects must have a coherent depth relative to the scene
- Lighting effects are visible in 3D so should be computed correctly
  - Highlight and specular are probably best looking evaluated with mono eye origin
  - Reflection and Refraction should be evaluated with stereo eyes
Pseudo 3D objects: Sky box, Billboards...

- Sky box should be drawn with a valid depth further than the regular scene
  - Must be Stereo Projected
  - Best is at a very Far distance so Parallax is maximum
  - And cover the full screen
- Billboard elements (Particles, leaves) should be rendered in a plane parallel to the viewing plane
  - Doesn’t look perfect
- Relief mapping looks bad
Several 3D scenes

- Different 3D scenes rendered in the same frame using different scales
  - Portrait viewport of selected character
  - Split screen
- Since scale of the scene is different, Must use a different Convergence to render each scene
Out of the screen objects

- The user’s brain is fighting against the perception of hovering objects out of the screen
  - Extra care must be taken to achieve a convincing effect
- Objects should not be clipped by the edges of the window
  - Be aware of the extra horizontal guard bands
- Move object slowly from inside the screen to the outside area to give eyes time to adapt
  - Make smooth visibility transitions
  - No blinking
- Realistic rendering helps
2D Objects

2D object in depth attached to 3D anchor point

Billboards in depth
Particles with 3D positions

Starcraft2 screenshot, Courtesy of Blizzard
2D Objects must be drawn at a valid Depth

- With no stereo projection
  - Head Up Display interface
  - UI elements
  - Either draw with no stereo projection or with stereo projection at Convergence

- At the correct depth when interacting with the 3D scene
  - Labels or billboards in the scene
  - Must be drawn with stereo projection
  - Use the depth of the 3D anchor point used to define the position in 2D window space

- Needs to modify the 2D ortho projection to take into account Stereo
2D to 3D conversion
shader function

float4 2Dto3DclipPosition(
    in float2 posClip : POSITION, // Input position in clip space
    uniform float depth // Depth where to draw the 2D object
) : POSITION // Output the position in clip space
{
    return float4(
        posClip.xy * depth, // Simply scale the posClip by the depth
        0, // Z is not used if the depth buffer is not used
        // If needed Z = (depth * f - nf)/(f - n);
        // ( For DirectX )
        depth // W is the Z in eye space
    );
}

Selection, Pointing in S3D

- Selection or pointing UI interacting with the 3D scene don’t work if drawn mono
  - Mouse Cursor at the pointed object’s depth
    Can not use the HW cursor
  - Crosshair
- Needs to modify the projection to take into account depth of pointed elements
  - Draw the UI as a 2D element in depth at the depth of the scene where pointed
  - Compute the depth from the Graphics Engine or eval on the fly from the depth buffer (Contact me for more info)
- Selection Rectangle is not perfect, could be improved
3D Objects Culling

When culling is done against the mono frustum...
3D Objects Culling

... Some in screen regions are missing in the right and left frustum ... They should be visible
3D Objects Culling

... And we don’t want to see out of the screen objects only in one eye ... It disturbs the stereo perception
3D Objects Culling

Here is the frustum we want to use for culling
3D Objects Culling
Computing Stereo Frustum origin offset

\[ Z = \text{Convergence} / (1 + 1 / \text{Separation}) \]
3D Objects Culling

- Culling this area is not always a good idea
- Blacking out pixels in this area is better
  - Through a shader

- Equivalent to the “Floating window” used in movies
Fetching Stereo Render Target

- When fetching from a stereo render target use the good texture coordinate
  - Render target is addressed in STEREO IMAGE SPACE
  - Use the pixel position provided in the pixel shader
  - Or use a texture coordinate computed in the vertex shader correctly
Unprojection in pixel shader

- When doing deferred shading technique, Pixel shader fetch the depth buffer (beware of the texcoord used, cf previous slide)
  - And evaluate a 3D clip position from the Depth fetched and XY viewport position
  - Make sure to use a Stereo Unprojection Inverse transformation to go to Mono Eye space
  - Otherwise you will be in a Stereo Eye Space !
One or two things to look at

WHAT’S NEXT?
Performance considerations

- At worse the frame rate is divided by 2
- But applications are rarely GPU bound so less expensive in practice
  - Since using Vsutch when running in stereo, you see the standard Vsync frequency jumps
- Not all the rendering is executed twice (Shadow maps)

- Memory is allocated twice for all the stereo surfaces
  - Try to reuse render targets when possible to save memory

- Get another GPU 😊
Tessellation

- Works great with stereoscopy
- Unigine Demo
Letterbox

- Emphasize the out of the screen effect
- Simply Draw 2 extra horizontal bands at Convergence
  - Out of the screen objects can overdraw the bands

G-Force movie from Walt Disney
Nvidia Demo Sled

SHOW TIME