Computer Graphics

Prof. Feng Liu

Fall 2015

http://www.cs.pdx.edu/~fliu/courses/cs447/

10/15/2015
Last time

☐ Filtering
☐ Resampling
Today

- Compositing
- NPR
- 3D Graphics Toolkits
  - Transformations
Compositing

- Compositing combines components from two or more images to make a new image
  - Special effects are easier to control when done in isolation
  - Even many all live-action sequences are more safely shot in different layers
Historically ...

- The basis for film special effects
  - Create digital imagery and composite it into live action
  - It was necessary for films (like Star Wars) where models were used
  - It was done with film and masks, and was time consuming and expensive

- Important part of animation - even hand animation
  - Background change more slowly than foregrounds, so composite foreground elements onto constant background
  - It was a major advance in animation - the *multiplane camera* first used in Snow White (1937)
Perfect Storm
Animated Example

over

=
A matte is an image that shows which parts of another image are foreground objects.

Term dates from film editing and cartoon production.

How would I use a matte to insert an object into a background?

How are mattes usually generated for television?
Working with Mattes

☐ To insert an object into a background
   ■ Call the image of the object the source
   ■ Put the background into the destination
   ■ For all the source pixels, if the matte is white, copy the pixel, otherwise leave it unchanged

☐ To generate mattes:
   ■ Use smart selection tools in Photoshop or similar
     ☐ They outline the object and convert the outline to a matte
   ■ Blue Screen: Photograph/film the object in front of a blue background, then consider all the blue pixels in the image to be the background
Compositing

- Compositing is the term for combining images, one over the other
  - Used to put special effects into live action
Alpha

- Basic idea: Encode opacity information in the image
- Add an extra channel, the *alpha* channel, to each image
  - For each pixel, store R, G, B and Alpha
  - alpha = 1 implies full opacity at a pixel
  - alpha = 0 implies completely clear pixels
- There are many interpretations of alpha
  - Is there anything in the image at that point (web graphics)
  - Transparency (real-time OpenGL)
- Images are now in RGBA format, and typically 32 bits per pixel (8 bits for alpha)
- All images in the project are in this format
Pre-Multiplied Alpha

- Instead of storing (R,G,B,\(\alpha\)), store (\(\alpha\)R,\(\alpha\)G,\(\alpha\)B,\(\alpha\))
- The compositing operations in the next several slides are easier with pre-multiplied alpha
- To display and do color conversions, must extract RGB by dividing out \(\alpha\)
  - \(\alpha=0\) is always black
  - Some loss of precision as \(\alpha\) gets small, but generally not a big problem
Compositing Assumptions

- We will combine two images, $f$ and $g$, to get a third composite image
- Both images are the same size and use the same color representation
- Multiple images can be combined in stages, operating on two at a time
Basic Compositing Operation

- At each pixel, combine the pixel data from $f$ and the pixel data from $g$ with the equation:

$$c_o = Fc_f + Gc_g$$

- $F$ and $G$ describe how much of each input image survives, and $c_f$ and $c_g$ are pre-multiplied pixels, and all four channels are calculated.

- To define a compositing operation, define $F$ and $G$.
Basic Compositing Operation

- $F$ and $G$ are simple functions of the alpha values

\[ c_o = F(\alpha_f, \alpha_g)c_f + G(\alpha_f, \alpha_g)c_g \]

- $F$ and $G$ are chosen (independently)

- Different choices give different operations

- To code it, you can write one compositor and give it 6 numbers (3 for $F$, 3 for $G$) to say which function
  - Constant of 0 or 1
  - $\alpha_f$ is multiplied by -1, 0 or 1. Similar for $\alpha_g$
Sample Images

Images

Alphas
Sample Images

Images

RGB

Alphas
Sample Images

Images

Pre-multiplied RGBA

<table>
<thead>
<tr>
<th>0,1,0 1</th>
<th>0,0.5,0 0.5</th>
<th>0, 0, 0 0</th>
<th>1,0,0 1</th>
<th>1,0,0 1</th>
<th>1,0,0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1,0 1</td>
<td>0,0.5,0 0.5</td>
<td>0, 0, 0 0</td>
<td>0.5,0,0 0.5</td>
<td>0.5,0,0 0.5</td>
<td>0.5,0,0 0.5</td>
</tr>
<tr>
<td>0,1,0 1</td>
<td>0,0.5,0 0.5</td>
<td>0, 0, 0 0</td>
<td>0, 0, 0 0</td>
<td>0, 0, 0 0</td>
<td>0, 0, 0 0</td>
</tr>
</tbody>
</table>

Alphas

18
“Over” Operator

- Computes composite with the rule that $f$ covers $g$

\[
F = 1
\]

\[
G = 1 - \alpha_f
\]

\[
c_o = Fc_f + Gc_g = c_f + (1 - \alpha_f)c_g
\]
“Over” Operator

\[ c_o = Fc_f + Gc_g = c_f + (1 - \alpha_f)c_g \]

<table>
<thead>
<tr>
<th>f</th>
<th>g</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1,0</td>
<td>0,0.5,0</td>
<td>0,0,0</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
<td>0</td>
</tr>
<tr>
<td>0,1,0</td>
<td>0,0.5,0</td>
<td>0,0,0</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
<td>0</td>
</tr>
<tr>
<td>0,1,0</td>
<td>0,0.5,0</td>
<td>0,0,0</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
<td>0</td>
</tr>
</tbody>
</table>

Over

\[ \begin{array}{cccc}
1,0,0 & 1,0,0 & 1,0,0 \\
1   & 1   & 1   \\
0.5,0,0 & 0.5,0,0 & 0.5,0,0 \\
0.5 & 0.5 & 0.5 \\
0,0,0 & 0,0,0 & 0,0,0 \\
0 & 0 & 0 \\
\end{array} \]

\[ \begin{array}{cccc}
0,1,0 & 0.5,0.5,0 & 1,0,0 \\
1   & 1   & 1   \\
0.25,0.5,0 & 0.75 & 0.5,0,0 \\
0.5 & 0.5 & 0.5 \\
0,0,0 & 0,0,0 & 0,0,0 \\
0 & 0 & 0 \\
\end{array} \]
## “Over” Operator: Extract RGB Color

Pre-multiplied RGBA

<table>
<thead>
<tr>
<th>RGBA</th>
<th>0.1,0</th>
<th>0.5,0.5,0</th>
<th>1,0,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0,0,1</td>
<td>0,1,0</td>
<td>0.5,0.5,0</td>
<td>1,0,0</td>
</tr>
<tr>
<td>0,1,0,1</td>
<td>0,1,0</td>
<td>0.25,0.5,0</td>
<td>0.5,0,0</td>
</tr>
<tr>
<td>0,1,0,1</td>
<td>0,0,0</td>
<td>0,0,0,0</td>
<td>0</td>
</tr>
</tbody>
</table>

### RGB

<table>
<thead>
<tr>
<th>RGBA</th>
<th>0.1,0</th>
<th>0.5,0.5,0</th>
<th>1,0,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0,0,1</td>
<td>0,1,0</td>
<td>0.5,0.5,0</td>
<td>1,0,0</td>
</tr>
<tr>
<td>0,1,0,1</td>
<td>0,1,0</td>
<td>0.25,0.5,0</td>
<td>0.5,0,0</td>
</tr>
<tr>
<td>0,1,0,1</td>
<td>0,0,0</td>
<td>0,0,0,0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Alpha

<table>
<thead>
<tr>
<th>RGBA</th>
<th>0.1,0</th>
<th>0.5,0.5,0</th>
<th>1,0,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0,0,1</td>
<td>0,1,0</td>
<td>0.5,0.5,0</td>
<td>1,0,0</td>
</tr>
<tr>
<td>0,1,0,1</td>
<td>0,1,0</td>
<td>0.25,0.5,0</td>
<td>0.5,0,0</td>
</tr>
<tr>
<td>0,1,0,1</td>
<td>0,0,0</td>
<td>0,0,0,0</td>
<td>0</td>
</tr>
</tbody>
</table>
“Over” Operator

☐ If there’s some $f$, get $f$, otherwise get $g$
“Inside” Operator

- Computes composite with the rule that only parts of $f$ that are inside $g$ contribute

\[
F = \alpha_g \\
G = 0
\]
**“Inside” Operator**

\[ c_o = F c_f + G c_g = \alpha_g c_f \]

<table>
<thead>
<tr>
<th>( f )</th>
<th>( g )</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0,1,0 )</td>
<td>( 1 )</td>
<td>( 0,1,0 )</td>
</tr>
<tr>
<td>( 0,0.5,0 )</td>
<td>( 0.5 )</td>
<td>( 0,0.5,0 )</td>
</tr>
<tr>
<td>( 0,0,0 )</td>
<td>( 0 )</td>
<td>( 0,0,0 )</td>
</tr>
<tr>
<td>( 0,1,0 )</td>
<td>( 0.5,0,0 )</td>
<td>( 0,0.5,0 )</td>
</tr>
<tr>
<td>( 0,0.5,0 )</td>
<td>( 0.5 )</td>
<td>( 0,0.5,0 )</td>
</tr>
<tr>
<td>( 0,0,0 )</td>
<td>( 0 )</td>
<td>( 0,0,0 )</td>
</tr>
<tr>
<td>( 0,1,0 )</td>
<td>( 0,0,0 )</td>
<td>( 0,0,0 )</td>
</tr>
<tr>
<td>( 0,0.5,0 )</td>
<td>( 0 )</td>
<td>( 0,0,0 )</td>
</tr>
<tr>
<td>( 0,0,0 )</td>
<td>( 0 )</td>
<td>( 0,0,0 )</td>
</tr>
</tbody>
</table>
“Inside” Operator

- Get $f$ to the extent that $g$ is there, otherwise nothing

\[
\text{inside} = \quad \text{inside} = \quad \text{inside}
\]
“Outside” Operator

1. Computes composite with the rule that only parts of $f$ that are outside $g$ contribute

\[
F = 1 - \alpha_g \\
G = 0
\]
“Outside” Operator

- Get $f$ to the extent that $g$ is \textbf{not} there, otherwise nothing

\[ \text{outside} = \text{result} \]
“Atop” Operator

- Computes composite with the over rule but restricted to places where there is some $g$

\[
F = \alpha_g
\]

\[
G = 1 - \alpha_f
\]
“Atop” Operator

Get $f$ to the extent that $g$ is there, otherwise $g$
“Xor” Operator

- Computes composite with the rule that $f$ contributes where there is no $g$, and $g$ contributes where there is no $f$

\[
F = 1 - \alpha_g \\
G = 1 - \alpha_f
\]
“Xor” Operator

- Get \( f \) to the extent that \( g \) is not there, and \( g \) to extent of no \( f \)

\[ \text{xor} = \]

\[ \]

\[ = \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]
“Clear” Operator

- Computes a clear composite
  \[ F = 0 \]
  \[ G = 0 \]

- Note that \((0,0,0,\alpha>0)\) is a partially opaque black pixel, whereas \((0,0,0,0)\) is fully transparent, and hence has no color
“Set” Operator

- Computes composite by setting it to equal $f$
  \[ F = 1 \]
  \[ G = 0 \]

- Copies $f$ into the composite
Compositing Operations

$F$ and $G$ describe how much of each input image survives, and $c_f$ and $c_g$ are pre-multiplied pixels, and all four channels are calculated

$$c_o = Fc_f + Gc_g$$

<table>
<thead>
<tr>
<th>Operation</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Over</td>
<td>1</td>
<td>$1 - \alpha_f$</td>
</tr>
<tr>
<td>Inside</td>
<td>$\alpha_g$</td>
<td>0</td>
</tr>
<tr>
<td>Outside</td>
<td>$1 - \alpha_g$</td>
<td>0</td>
</tr>
<tr>
<td>Atop</td>
<td>$\alpha_g$</td>
<td>$1 - \alpha_f$</td>
</tr>
<tr>
<td>Xor</td>
<td>$1 - \alpha_g$</td>
<td>$1 - \alpha_f$</td>
</tr>
<tr>
<td>Clear</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Set</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Unary Operators

- **Darken**: Makes an image darker (or lighter) without affecting its opacity

  $$darken(f, \phi) \equiv (\phi r_f, \phi g_f, \phi b_f, \alpha_f)$$

- **Dissolve**: Makes an image transparent without affecting its color

  $$dissolve(f, \delta) \equiv (\delta r_f, \delta g_f, \delta b_f, \delta \alpha_f)$$
“PLUS” Operator

- Computes composite by simply adding $f$ and $g$, with no overlap rules

$$C_o = C_f + C_g$$

- Useful for defining *cross-dissolve* in terms of compositing:

$$\text{cross}(f, g, t) = \text{dissolve}(f, t) \text{ plus disslove}(g, 1-t)$$
Obtaining $\alpha$ Values

- Hand generate (paint a grayscale image)
- Automatically create by segmenting an image into foreground background:
  - Blue-screening is the analog method
  - Remarkably complex to get right
  - “Lasso” is the Photoshop operation
- With synthetic imagery, use a special background color that does not occur in the foreground
  - Brightest blue or green is common
Compositing With Depth

- Can store pixel “depth” instead of alpha
- Then, compositing can truly take into account foreground and background
- Generally only possible with synthetic imagery
  - Image Based Rendering is an area of graphics that, in part, tries to composite photographs taking into account depth
Today

- More Compositing
- Non-photorealistic Rendering (NPR)
- 3D Graphics Toolkits
  - Transformations
Many methods have been proposed to make a photo look like a painting.

Today we look at one: *Painterly-Rendering with Brushes of Multiple Sizes*.

Basic ideas:
- Build painting one layer at a time, from biggest to smallest brushes.
- At each layer, add detail missing from previous layer.

*Aaron Hertzmann. Painterly rendering with curved brush strokes of multiple sizes, SIGGRAPH 1998*
Algorithm 1

```plaintext
function paint(sourceImage, R_1 \ldots R_n) // take source and several brush sizes
{
    canvas := a new constant color image
    // paint the canvas with decreasing sized brushes
    for each brush radius R_i, from largest to smallest do
    {
        // Apply Gaussian smoothing with a filter of size const * radius
        // Brush is intended to catch features at this scale
        referenceImage = sourceImage * G(fs R_i)
        // Paint a layer
        paintLayer(canvas, referenceImage, R_i)
    }
    return canvas
}
```
Algorithm 2

procedure paintLayer(canvas, referenceImage, R) // Add a layer of strokes
{
    S := a new set of strokes, initially empty
    D := difference(canvas, referenceImage) // euclidean distance at every pixel
    for x=0 to imageWidth stepsize grid do // step in size that depends on brush radius
        for y=0 to imageHeight stepsize grid do {
            // sum the error near (x,y)
            M := the region (x-grid/2..x+grid/2, y-grid/2..y+grid/2)
            areaError := sum(D_{i,j} for i,j in M) / grid^2
            if (areaError > T) then {
                // find the largest error point
                (x1,y1) := max D_{i,j} in M
                s := makeStroke(R, x1,y1, referenceImage)
                add s to S
            }
        }
    paint all strokes in S on the canvas, in random order
}
Point Style

- Uses round brushes
- We provide a routine to “paint” round brush strokes into an image for the project
Results

Original

Biggest brush

Medium brush added

Finest brush added
Where to now...

- We are now done with images
- We will spend several weeks on the mechanics of 3D graphics
  - Coordinate systems and Viewing
  - Clipping
  - Drawing lines and polygons
  - Lighting and shading
Graphics Toolkits

- Graphics toolkits typically take care of the details of producing images from geometry
- Input (via API functions):
  - Where the objects are located and what they look like
  - Where the camera is and how it behaves
  - Parameters for controlling the rendering
- Functions (via API):
  - Perform well defined operations based on the input environment
- Output: Pixel data in a framebuffer - an image in a special part of memory
  - Data can be put on the screen
  - Data can be read back for processing (part of toolkit)
OpenGL

- OpenGL is an open standard graphics toolkit
  - Derived from SGI’s GL toolkit
- Provides a range of functions for modeling, rendering and manipulating the framebuffer
- What makes a good toolkit?
- Alternatives: Direct3D, Java3D - more complex and less well supported
A Good Toolkit...

- Everything is a trade-off
- Functionality
  - Compact: a minimal set of commands
  - Orthogonal: commands do different things and can be combined in a consistent way
  - Speed
- Ease-of-Use and Documentation
- Portability
- Extensibility
- Standards and ownership
- Not an exhaustive list ...
Coordinate Systems

- The use of *coordinate systems* is fundamental to computer graphics.
- Coordinate systems are used to describe the locations of points in space, and directions in space.
- Multiple coordinate systems make graphics algorithms easier to understand and implement.
Coordinate Systems (2)

- Different coordinate systems represent the same point in different ways

\[ \begin{align*}
(2,3) \\
(1,2)
\end{align*} \]
Transformations

Transformations convert points between coordinate systems

\[ u = x - 1 \]
\[ v = y - 1 \]

\[ x = u + 1 \]
\[ y = v + 1 \]
Transformations (Alternate Interpretation)

- Transformations modify an object’s shape and location in one coordinate system

- The previous interpretation is better for some problems, this one is better for others

\[
x' = x - 1 \\
y' = y - 1
\]

\[
x = x' + 1 \\
y = y' + 1
\]
2D Translation

- Moves an object

\[
\begin{bmatrix}
  x' \\
  y'
\end{bmatrix} = \begin{bmatrix}
  ? & ? \\
  ? & ?
\end{bmatrix} \begin{bmatrix}
  x \\
  y
\end{bmatrix} + \begin{bmatrix}
  ? \\
  ?
\end{bmatrix}
\]
2D Translation

Moves an object

\[
\begin{bmatrix}
  x' \\
  y'
\end{bmatrix} = \begin{bmatrix}
  1 & 0 \\
  0 & 1
\end{bmatrix}\begin{bmatrix}
  x \\
  y
\end{bmatrix} + \begin{bmatrix}
  b_x \\
  b_y
\end{bmatrix}
\]
2D Scaling

- Resizes an object in each dimension

\[
\begin{bmatrix}
x' \\
y'
\end{bmatrix} = \begin{bmatrix}
? & ? \\
? & ?
\end{bmatrix}\begin{bmatrix}
x \\
y
\end{bmatrix} + \begin{bmatrix}
? \\
?
\end{bmatrix}
\]
2D Scaling

- Resizes an object in each dimension

\[
\begin{bmatrix}
    x' \\
y'
\end{bmatrix} = \begin{bmatrix}
s_x & 0 \\
0 & s_y
\end{bmatrix} \begin{bmatrix}
x \\
y
\end{bmatrix} + \begin{bmatrix}
0 \\
0
\end{bmatrix}
\]
2D Rotation

- Rotate counter-clockwise about the origin by an angle $\theta$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} ? & ? \\ ? & ? \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} ? \end{bmatrix}$$
2D Rotation

- Rotate counter-clockwise about the origin by an angle $\theta$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$
X-Axis Shear

Shear along x axis (What is the matrix for y axis shear?)

\[
\begin{bmatrix}
x' \\
y'
\end{bmatrix} = \begin{bmatrix}
? & ? \\
? & ?
\end{bmatrix} \begin{bmatrix}
x
\end{bmatrix} + \begin{bmatrix}
?
\end{bmatrix}
\]
X-Axis Shear

Shear along x axis (What is the matrix for y axis shear?)

\[
\begin{bmatrix}
  x' \\
  y'
\end{bmatrix}
= \begin{bmatrix}
  1 & sh_x \\
  0 & 1
\end{bmatrix}
\begin{bmatrix}
  x \\
  y
\end{bmatrix}
+ \begin{bmatrix}
  0 \\
  0
\end{bmatrix}
\]
Reflect About X Axis

\[
\begin{bmatrix}
x'
\end{bmatrix} = \begin{bmatrix}
? & ?
\end{bmatrix} \begin{bmatrix}
x
\end{bmatrix} + \begin{bmatrix}
?
\end{bmatrix}
\]
Reflect About X Axis

\[
\begin{bmatrix}
x' \\
y'
\end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}
\]
2D Affine Transformations

- An affine transformation is one that can be written in the form:

\[ \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a_{xx} & a_{xy} \\ a_{yx} & a_{yy} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_x \\ b_y \end{bmatrix} \]

or

\[ x' = a_{xx}x + a_{xy}y + b_x \]
\[ y' = a_{yx}x + a_{yy}y + b_y \]
Composition of Affine Transforms

Any affine transformation can be composed as a sequence of simple transformations:

- Translation
- Scaling (possibly with negative values)
- Rotation
Rotating About An Arbitrary Point

What happens when you rotate an object about an arbitrary point that is not the origin?
Rotating About An Arbitrary Point

What happens when you rotate an object about an arbitrary point that is not the origin?
How Do We Compute It?

How do we rotate an about an arbitrary point?

- Hint: we know how to rotate about the origin of a coordinate system
Rotating About An Arbitrary Point

\[(a, b) \quad (0, 0)\]
Rotating About An Arbitrary Point

- Say you wish to rotate about the point \((a,b)\)
- You know how to rotate about \((0,0)\)
- Translate so that \((a,b)\) is at \((0,0)\)
  - \(x' = x - a, \ y' = y - b\)
- Rotate
  - \(x'' = (x-a) \cos \theta - (y-b) \sin \theta, \ y'' = (x-a) \sin \theta + (y-b) \cos \theta\)
- Translate back again
  - \(x_f = x'' + a, \ y_f = y'' + b\)
Rotating About An Arbitrary Point

- Say $R$ is the rotation matrix to apply, and $p$ is the point about which to rotate
- Translation to Origin: $x' = x - p$
- Rotation: $x'' = Rx' = R(x - p) = Rx - Rp$
- Translate back: $x''' = x'' + p = Rx + (-Rp + p)$
- The translation component of the composite transformation involves the rotation matrix. What a mess!
Next Time

☐ Composing transformations
☐ 3D Transformations
☐ Viewing
☐