Basics of rendering - Algorithms with WPF

Gaëtan Blaise-Cazalet - Head of programming section

Basics of Rendering - 2d

Why WPF ?

- We want a easy to use tool to test 2d (and 3d) rendering algorithms and concepts
- A simple tool for simple ideas. We will use more complex tools further in the year.
- WPF uses C# (and XAML). C# is a widely used language, and is a the main language for Unity scripts.
- It is an efficient tool to create windows application, compatible with older versions of windows.

- Nevertheless : Microsoft wants to replace WPF by UWP. WPF years are counted.
  (link)
- WPF uses an old version of DirectX (DirectX 9).

Historical vocabulary

- Old painting applications, that would paint pixels on rectangular canvas, are called raster graphics applications
- The fact to convert something to pixels is called rasterization
- Those applications were drawing primitives, ie shapes that have a brush (inside the shape) and pen (edge of the shape)
- Two modes of rendering :
  - Immediate mode when the application regenerate a set of primitives to update a scene.
  - Retained mode when the content of the scene is represented as a scene graph. Updating the scene graph make the application update the rendering

Let's start with WPF

- Create a new WPF application in Visual Studio. It starts with a window.
- Create a canvas that fill the window
- We want to draw a clock.
- Add an Ellipse to figure your clock background

- MainWindow.xaml

```
<Window x:Class="_01.Canvas.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:_01.Canvas"
        mc:Ignorable="d"
        Title="Intro to rendering" Height="600" Width="800">
    <Canvas ClipToBounds="False">
        <Ellipse Canvas.Left="-10" Canvas.Top="-10" Width="20" Height="20" Fill="LightGray" />
    </Canvas>
</Window>
```

Using transforms

- We want to keep our coordinate system on 100 units, but we also want the clock to fill the window
- We'll use transforms to adapt our drawing

- MainWindow.xaml

```
<Canvas ClipToBounds="False">
   <Ellipse Canvas.Left="-10" Canvas.Top="-10" Width="20" Height="20" Fill="LightGray" />

   <Canvas.RenderTransform>
      <TransformGroup>
         <ScaleTransform ScaleX="4.8" ScaleY="4.8" CenterX="0" CenterY="0" />
         <TranslateTransform X="48" Y="48" />
      </TransformGroup>
   </Canvas.RenderTransform>
</Canvas>
```

Use a template as a clock-hand

- We want to reuse the clock-hand
- We use a control template for that

- MainWindow.xaml

```
<Canvas ClipToBounds="False">
   <Canvas.Resources>
      <ControlTemplate x:Key="ClockHandTemplate">
         <Polygon Points="-0.3,-1 0.2,8 0,9 0.2,8 0.3,1" Fill="Navy" />
      </ControlTemplate>
   </Canvas.Resources>

   <Ellipse Canvas.Left="-10" Canvas.Top="-10" Width="20" Height="20" Fill="LightGray" />
   <Control Name="MinuteHand" Template="{StaticResource ClockHandTemplate}" />

   <Canvas.RenderTransform>
      <TransformGroup>
         <ScaleTransform ScaleX="4.8" ScaleY="4.8" CenterX="0" CenterY="0" />
         <TranslateTransform X="48" Y="48" />
      </TransformGroup>
   </Canvas.RenderTransform>
</Canvas>
```

The hour clock-hand

- MainWindow.xaml

```
...
<Control Name="MinuteHand" Template="{StaticResource ClockHandTemplate}" />
<Control Name="HourHand" Template="{StaticResource ClockHandTemplate}">
   <Control.RenderTransform>
      <TransformGroup>
         <ScaleTransform ScaleX="1.7" ScaleY="0.7" CenterX="0" CenterY="0" />
         <RotateTransform Angle="45" />
      </TransformGroup>
   </Control.RenderTransform>
</Control>
...
```

- Now add a red thin clock-hand for the seconds

Add animation

- Set the hour handle to top (12)
- We add an additional rotation with a name (ActualTimeHour)
- We set an animation for this variable
- MainWindow.xaml

```
<Control Name="HourHand" Template="{StaticResource ClockHandTemplate}">
    <Control.RenderTransform>
        <TransformGroup>
            <ScaleTransform ScaleX="1.7" ScaleY="0.7" CenterX="0" CenterY="0" />
            <RotateTransform Angle="180" />
            <RotateTransform x:Name="ActualTimeHour" Angle="0" />
        </TransformGroup>
    </Control.RenderTransform>
</Control>

<Canvas.Triggers>
    <EventTrigger RoutedEvent="FrameworkElement.Loaded">
        <BeginStoryboard>
            <Storyboard>
                <DoubleAnimation Storyboard.TargetName="ActualTimeHour" Storyboard.TargetProperty="Angle"
                        From="0" To="360" Duration="00:00:10.00" RepeatBehavior="Forever" />
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</Canvas.Triggers>
```

- Make every handle turn
- Make the clock turn at real time rate
- Find a way to initialize the clock at correct hour

Exercice - The covered wagon

Your goal: to modify the clock application to make and animate a 2D model of a cart (viewed from the side), consisting of a rectangle resting atop two wheels. Each wheel should have four spokes constructed via instantiations of the clock-hand template that we've already defined for you. Feel free to rename things to match the new semantics, e.g. "SpokeStencil" instead of "ClockHandStencil".

1. First build one wheel, complete with the four spokes, but don't worry about rotating it yet. It's preferable to use a coordinate system that is comfortable for the modeling process, i.e. with the center of the wheel at (0,0).

2. Wrap the entire wheel model up in a canvas and make it be a new template resource, as was previously done for the clock hand. You will end up with a structure like that shown below. To make sure you didn't damage its functionality, test by instantiating it onto the visible canvas.

- MainWindow.xaml

```
<Canvas.Resources>
 <ControlTemplate x:Key='SpokeStencil'>
  <Polygon ... />
 </ControlTemplate>

 <ControlTemplate x:Key='WheelStencil'>
  <Canvas>
   <Ellipse Width='2.0' ... />
   <Control Name='Spoke1'
          Template='{StaticResource SpokeStencil}'>

    <Control.RenderTransform> ...
   </Control>

   ... Three more spoke declarations will be here ...

  </Canvas>
 </ControlTemplate>
```

3. Make the wheel rotatable by attaching a transform to the element that is instantiating it, via the same technique that was used to make the minute hand rotatable.

4. Modify the animation element that originally rotated the minute hand, to make it rotate the wheel instead.

5. Add to the scene a rectangle and another instance of the wheel template to make a simple model of a cart or wagon. Of course, the second wheel instance must be translated differently to ensure it doesn't simply lie right on top of its sibling. Make sure the new wheel is separately animatable, and create a new animation element to handle its rotation.

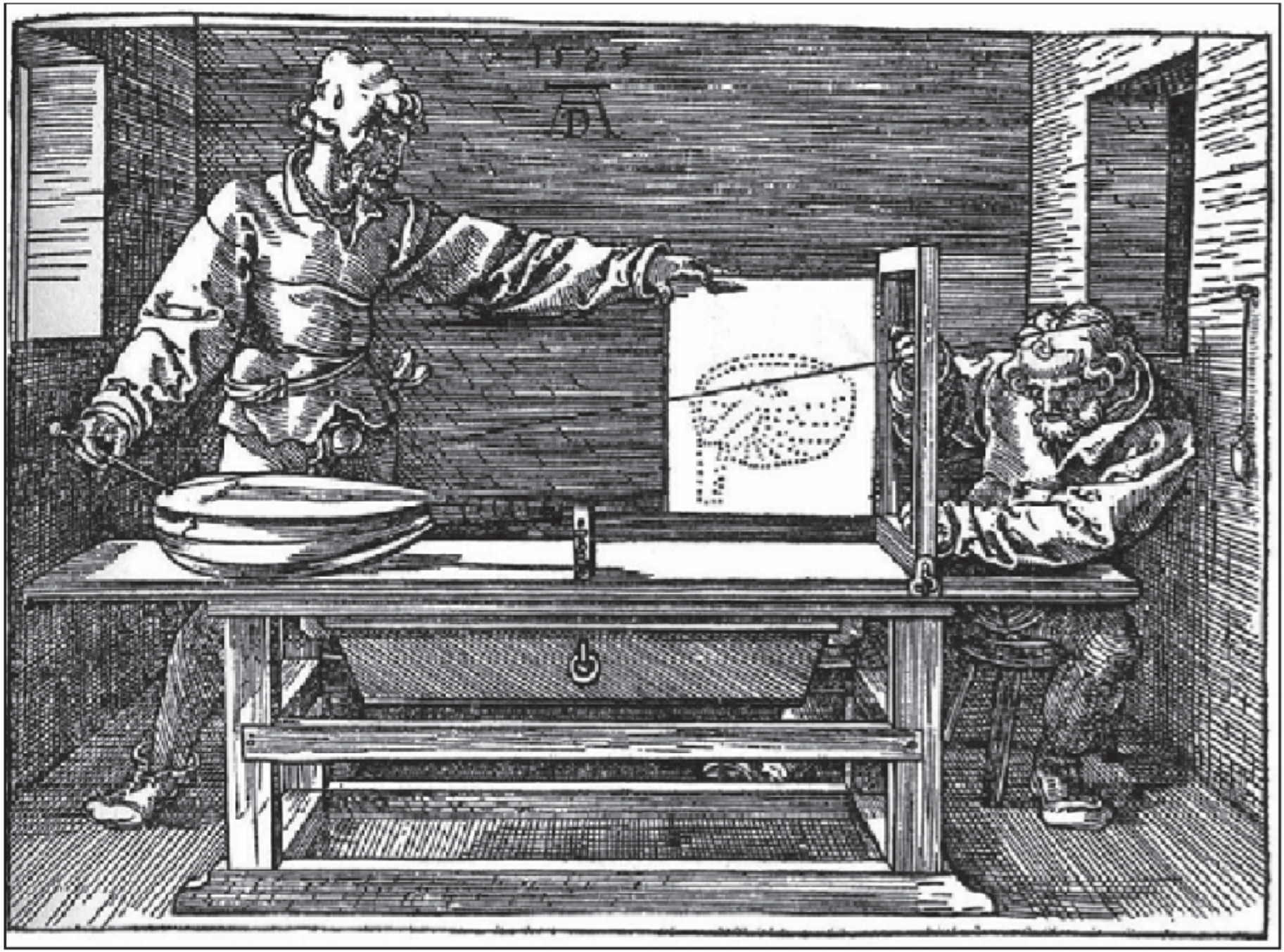
6. It's rather unrealistic for a cart with two spinning wheels to appear to be standing still! Use a canvas element to wrap up the entire model of the cart and attach a translate transform to this new canvas element; this transform will thus control the entire cart model as a whole. Connect an animator element to the translate transform's x value, to achieve the goal of having the entire model move horizontally to simulate the effect of the spinning wheels.

7. If you've completed the above step successfully, your cart probably zips off the screen the moment the movie begins, leaving your audience with nothing to be awed by! Set the AutoReverse property on all of the animators to simulate a cart that simply goes back and forth on a short track, thus staying on screen throughout the movie.

Projecting 3D to 2D

The Dürer's algorithm

Dürer's algorithm



Dürer's algorithm

- In a more algorithmic way:

```
Input: a scene containing some objects, location of eye-point
Output: a drawing of the objects

initialize drawing to be blank
foreach object o
   foreach visible point P of o
      Open shutter
      Place pointer at P
      if string from P to eye-point touches boundary of frame
         Do nothing
      else
         Hold a pencil at point where string passes through frame
         Hold string aside Close shutter to make pencil-mark on paper
         Release string
```

- Remarks :

1. This is done on visible points
2. There could be an infinite number of visible points
3. We handle the "touching / outside the frame" case. This is called "clipping"

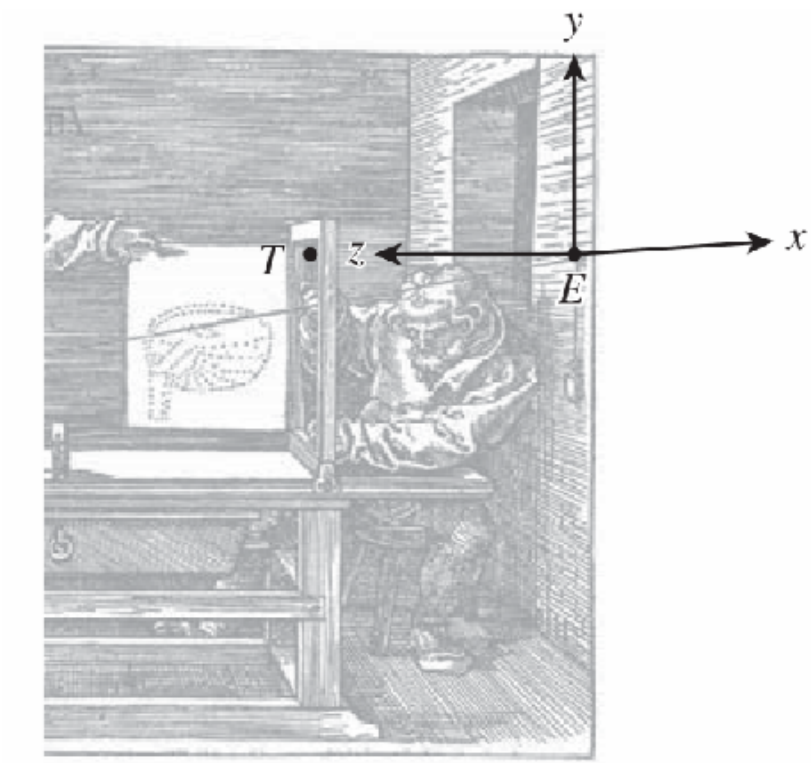
An other Dürer's algorithm

- Now, imagine the man has a graph paper (with squares)
- Now when the man holding the string touches the paper and the object, the painter look at the light on the point of the object
- If this point of the object is dark, the painter fills the square with black. If it is light, the painter let it white. If shady, the painter fills with some grey nuance.
- This "per-pixel" approach is the essence of ray-tracing!

The first algorithm is faster if the shape to draw is simple. The second if the scene is complex.

Choosing a geometry

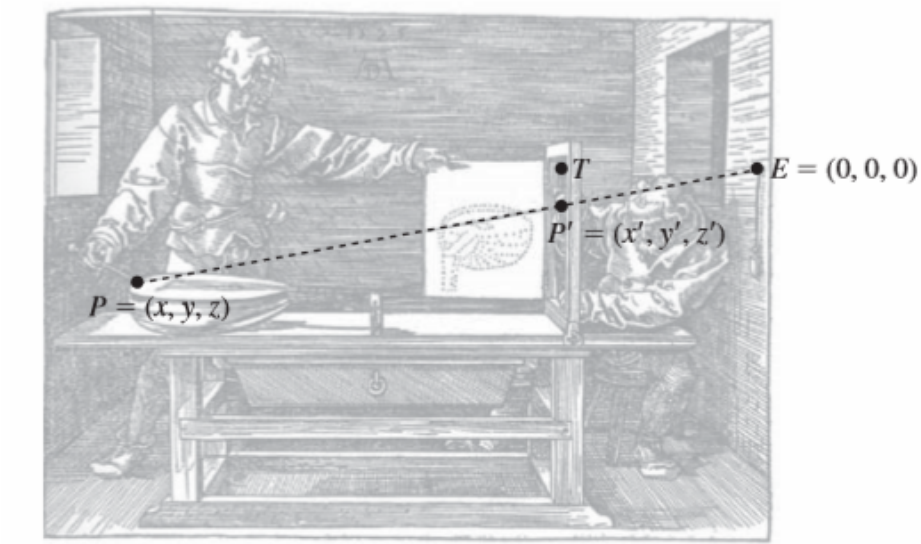
- We want to draw a cube, defined by its vertices (plural of vertex). We'll draw a wireframe cube (only edges).
- We use this coordinates system :



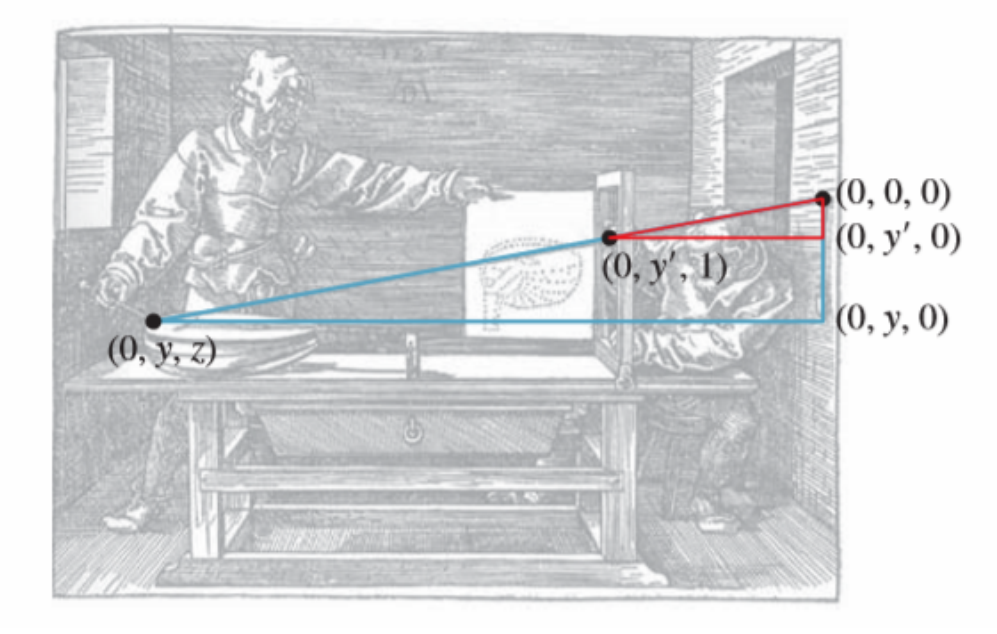
- (0, 0, 0) is the E point, on the wall
- The point from E along the z axis, on the frame is called T. It is at (0, 0, 1)
- z is depth, y is vertical, x is horizontal, along the plane of the drawing and of the wall
- The drawing extends from (Xmin, Ymin, 1) to (Xmax, Ymax, 1)

Objective

- We want to find P' coordinates (x', y', z'), a projection of the object point P (x, y, z) on the paper
- We know that z' = 1



Projection



- We choose to work on the x = 0 plane
- P' is (0, y', 1) P is (0, y, z)
- Red rectangle and blue rectangles are similar : y' / 1 = y / z
- With a different set of triangles, x' / 1 = x / z
- So x' = x/z and y' = y/z
- We want x to increase on the right, so we use -x
- Our algorithm is now :

```
Input: a scene containing some objects, location of eye-point
Output: a drawing of the objects

initialize drawing to be blank
foreach object o
    foreach visible point P of o
        if xmin ≤ (x/z) ≤ xmax and ymin ≤ (y/z) ≤ ymax
            make a point on the drawing at location (-x/z,y/z)
```

Draw algorithm

- We want to draw the cubes vertices and its edges :

Index	Coordinates
0	(−0.5, −0.5, 2.5)
1	(−0.5, 0.5, 2.5)
2	(0.5, 0.5, 2.5)
3	(0.5, −0.5, 2.5)
4	(−0.5, −0.5, 3.5)
5	(0.5, 0.5, 3.5)
6	(0.5, 0.5, 3.5)
7	(0.5, −0.5, 3.5)

Index	Endpoints
0	(0, 1)
1	(1, 2)
2	(2, 3)
3	(3, 0)
4	(0, 4)
5	(1, 5)
6	(2, 6)
7	(3, 7)
8	(4, 5)
9	(5, 6)
10	(6, 7)
11	(7, 4)

- We have to choose if we iterate through the edges, compute endpoints and draw a line, or if we iterate through the vertices and then create and draw edges. It is a matter of optimization, and depends the hardware we use. We choose the second way for now.
- We also have to consider what will happen if we draw an edge with a point inside the frame and a point outside. For now, we will let this problem to WPF.
- Our algorithm becomes :

```
Input: a scene containing one object ob
Output: a drawing of the objects

initialize drawing to be blank;
for (int i = 0; i < number of vertices in ob; i++) {
    Point3D P = vertices[i];
    pictureVertices[i] = Point(-P.x/P.z, P.y/P.z);
}
for {int i = 0; i < number of edges in ob; i++) {
    int i0 = edges[i][0];
    int i1 = edges[i][1];
    Draw a line segment from pictureVertices[i0] to pictureVertices[i1];
}
```

- Finally, we would like our screen coordinate to go from 0 to 1 ans not from Xmin to Xmax / Ymin to Ymax
- We can normalize : Xnew = (x - Xmin) / (Xmax - Xmin) Ynew = (y - Ymin) / (Ymax - Ymin)
- Because we negated x, values will be between -1 and 0.. So we have to add 1.

Final algorithm

- Here is the final algorithm :

```
Input: a scene containing one object ob
Output: a drawing of the objects

initialize drawing to be blank;
for (int i = 0; i < number of vertices in ob; i++) {
    Point3D P = vertices[i];
    double x = P.x/P.z;
    double y = P.y/P.z;
    pictureVertices[i] = Point(1 - (x - xmin)/(xmax - xmin), (y - ymin)/(ymax - ymin));
}
for {int i = 0; i < number of edges in ob; i++) {
    int i0 = edges[i][0];
    int i1 = edges[i][1];
    Draw a line segment from pictureVertices[i0] to pictureVertices[i1];
}
```

- We will implement in the TestApp.

Setting up the test app

- Copy the Visual Studio solution from the 00.start folder
- Create a new project called Durer
- Copy the content of Testbed2DApp project to the Durer project
- Add a reference to the TestBed2D project in the Durer project
- Erase the MainWindow files
- Set the Durer project as startup project and test it

Code

- Code the algorithm in Windows1.xaml.cs !

Exercices

1. The dots at the corners of the rendered cube appear behind the edges, which doesn't look all that natural; alter the program to draw the dots after the segments so that it looks better. Alter it again to not draw the dots at all, and to draw only the segments.

2. We can represent a shape by faces instead of edges; the cube in the Dürer program, for instance, might be represented by six square faces rather than the cube's 12 edges. We could then choose to draw a face only if it faces toward the eye.

"Drawing," in this case, might consist of just drawing the edges of the face. The result is a rendering of a wire-frame object, but with only the visible faces shown. If the object is convex, the rendering is correct; if it's not, then one face may partly obscure another. For a convex shape like a cube, with the property that the first two edges of any face are not parallel, it's fairly easy to determine whether a face with vertices (P0,P1,P2,...) is visible: You compute the cross product w =(P2−P1)x(P1−P0) of the vectors, and compare it to the vector v from the eye, E, to P0, that is,v = P0−E. If the dot product ofvandwis negative, the face is visible. This rule relies on ordering the vertices of each face so that the cross product w is a vector that, if it were placed at the face's center, would point into free space rather than into the object.

(a) Write down a list of faces for the cube, being careful to order them so that the computed "normal vector" w for each face points outward.

(b) Adjust your program to compute visibility for each face, and draw only the visible faces.

The cross- and dot-product based approach to visibility determination described in this exercise fails for more complex shapes, but later we'll see more sophisticated methods for determining whether a face is visible.

3. As in the preceding exercise, we can alter a wire-frame drawing to indicate front and back objects in other ways. We can, for instance, consider all the lines of the object (the edges of the cube, in our example) and sort them from back to front. If two line segments do not cross (as seen from the viewpoint) then we can draw them in either order. If they do cross, we draw the one farther from the eye first.

Furthermore, to draw a line segment (in black on a white background), we first draw a thicker version of the line segment in white, and then the segment itself in black at ordinary thickness. The result is that nearer lines "cross over and hide" farther lines.

(a) Draw an example of this on paper, using an eraser to simulate laying down the wide white strip.

(b) Think about how the lines will appear at their endpoints — will the white strips cause problems?

(c) Suppose two lines meet at a vertex but not at any other point. Does the order in which they're drawn matter? This "haloed line" approach to creating wire-frame images that indicate depth was used in early graphics systems, when drawing filled polygons was slow and expensive, and even later to help show internal structures of objects.

4. Create several simple models, such as a triangular prism, a tetrahedron, and a 1x2x3 box, and experiment with them in the rendering program.