Assignment 2

Michael Kazhdan

(601.457/657)
Outline

• The Util::Polynomial< Dim, Degree > class
• The Shape::processFirstIntersection method
Util::Polynomial< Dim, Degree >

This is a templated class, parameterized by:

- **Dim**: The dimension of the input space (i.e. the number of variables)
- **Degree**: The degree of the polynomial
Suppose you would like to define a polynomial describing a circle with radius 2, centered about the point (1,2) in 2D:

\[ P(x, y) = (x - 1)^2 + (y - 2)^2 - 4 \]
\[ = x^2 + y^2 - 2x - 4y + 1 \]
Util::Polynomial< Dim, Degree >

\[ P(x, y) = x^2 + y^2 - 2x - 4y + 1 \]

Defining the polynomial:

// Declare a polynomial in two variables of degree 2
Util::Polynomial2D< 2 > poly2D;

// Set the (non-zero) coefficients
poly2D.coefficient(2u,0u) = 1;    // the \( x^2 \) term
poly2D.coefficient(0u,2u) = 1;    // the \( y^2 \) term
poly2D.coefficient(1u,0u) = -2;   // the \( x \) term
poly2D.coefficient(0u,1u) = -4;   // the \( y \) term
poly2D.coefficient(0u,0u) = 1;    // the \( 1 \) term
Util::Polynomial< Dim, Degree >

\[ P(x, y) = x^2 + y^2 - 2x - 4y + 1 \]

Evaluating the polynomial:

```cpp
// The x and y coordinates at which to evaluate P
double x, y;

...// The point at which you want to evaluate P
Util::Point2D p(x, y);

...// Evaluate P at the prescribed point
double value1 = poly2D(p);
double value2 = poly2D(x, y);
```
\textbf{Util::Polynomial< Dim , Degree >}

\[ P(x, y) = x^2 + y^2 - 2x - 4y + 1 \]

Restricting the polynomial:

// The ray to which you want to restrict P
Util::Ray2D ray;

...

// The restriction of P to the ray is a polynomial
// of degree 2 in one variable
Util::Polynomial1D< 2 > poly1D = poly2D( ray );
Util::Polynomial< Dim, Degree >

\[ P(x, y) = x^2 + y^2 - 2x - 4y + 1 \]

Find the roots (of a 1D polynomial):

```cpp
Util::Polynomial1D< 2 > poly1D;
...
double roots[2];
unsigned int rootNum = p.roots( roots );
```

The array `roots` has to be large enough to store the roots of the polynomial (i.e. of size at least) `Degree`.

The number of roots may be less than the degree of the polynomial, so check the value of `rootNum`. 
Util::Polynomial< Dim, Degree >

\[ P(x, y) = x^2 + y^2 - 2x - 4y + 1 \]

Differentiating the polynomial:

// The partial derivative with respect to x
Util::Polynomial2D< 1 > dx = poly2D.d( 0 );

// The partial derivative with respect to y
Util::Polynomial2D< 1 > dy = poly2D.d( 1 );

Note that the degree of the derivative of a (partial) derivative of a polynomial is always one less than the degree of the polynomial.
Outline

• The Util::Polynomial< Dim, Degree > class
• The Shape::processFirstIntersection method
Ray-Casting:

Given a Shape, we would like to compute the first intersection with a Util::Ray3D.
Ray-Casting:

Given a Shape, we would like to compute the first intersection with a Util::Ray3D.

- **ray**: The Util::Ray3D to intersect with, in local coordinates. (The transformation into local coordinates is taken care of for you in AffineShape::processFirstIntersection)
Ray-Casting:

Given a Shape, we would like to compute the first intersection with a Util::Ray3D.

- **range**: The range, \([start, end]\) of values over which we are interested in an intersection.
  (If the ray intersects the shape at a time \(t\) outside the range, we do not consider that as a candidate intersection.)

Example:

- For shadow testing with point/spot lights, we can use this to restrict to surfaces between the ray source and the light.
- Can use this to discard intersections too close to the start of the ray (so we don’t keep intersecting the same point).
Ray-Casting:

Given a Shape, we would like to compute the first intersection with a Util::Ray3D.

- **rFilter**: An object of type `std::function<bool(double)>` that can be used to, more generally, filter out a candidate intersection as a function of the time $t$.

**Example:**
- When considering the intersection of shapes $A$ and $B$, we could consider:
  - Intersections with $A$, only if they are inside of $B$.
  - Intersections with $B$, only if they are inside of $A$.
Ray-Casting:

Given a Shape, we would like to compute the first intersection with a Util::Ray3D.

- **rKernel**: An object of type
  
  `std::function< void ( const ShapeProcessingInfo & ,
                             const RayShapeIntersectionInfo & ) >`
  
  that is to be invoked with the information about the first ray-shape intersection (`RayShapeIntersectionInfo`) and the accumulated transformations (`ShapeProcessingInfo`).

**Example:**

- This can be used to update/set the color of a pixel.
### Ray-Casting:

Given a `Shape`, we would like to compute the first intersection with a `Util::Ray3D`.

- **spInfo**: An object storing the global-to-local and local-to-global transformations needed to transform points, directions, and normals.
  
  (The updates of the transformations is taken care of for you in `AffineShape::processFirstIntersection`)
Ray-Casting:

Given a Shape, we would like to compute the first intersection with a Util::Ray3D.

- \( t\text{Idx} \): If you are implementing the ray-tracer as multi-threaded application, this integer gives the index of the thread responsible for processing this ray-shape intersection.

Example:

- You may opt to pre-allocate temporary arrays for computation so as not to have to do it on-the-fly. To avoid multiple threads writing to the same array, you can allocate a separate array for each thread and use \( t\text{Idx} \) to determine which array this thread should be writing into.
Shape::processFirstIntersection

Challenges:

To invoke rKernel on the first intersection with a ShapeList, we do not want to invoke rKernel on each of the child Shapes.

Example:

- If rKernel sets the pixel color, we would be setting the color with the first intersection of the last intersected child shape, not the first intersection.
Solution:

- Create a local variable to track the information about the first intersection.
**Shape::processFirstIntersection**

**Solution:**

- Create a local variable to track the information about the first intersection
- Define a new kernel to set the variable if a closer intersection was found

```cpp
class ShapeList : public Shape {
    ...
    std::vector< Shape * > shapes;
    ...
};

bool ShapeList::processFirstIntersection( ... )
{
    RayShapeIntersectionInfo iInfo;
    RayIntersectionKernel k = [&iInfo]( ... )
    {
        ...
    };
    for( const Shape *s : shapes )
        s->processFirstIntersection( ... , k , ... );
    rKernel( ... , iInfo );
};
```
Shape::processFirstIntersection

Solution:

- Create a local variable to track the information about the first intersection
- Define a new kernel to set the variable if a closer intersection was found
- Process the first intersections of the ray with the children using the new kernel

```cpp
class ShapeList : public Shape {
    ...
    std::vector< Shape * > shapes;
    ...
};

bool ShapeList::processFirstIntersection( ... )
{
    RayShapeIntersectionInfo iInfo;
    RayIntersectionKernel k = [&iInfo]( ... )
    {
        ...
    };
    for( const Shape *s : shapes )
        s->processFirstIntersection( ... , k , ... );
    rKernel( ... , iInfo );
};
```
Solution:

- Create a local variable to track the information about the first intersection
- Define a new kernel to set the variable if a closer intersection was found
- Process the first intersections of the ray with the children using the new kernel
- Invoke the original kernel with the information about the closest intersection