ITK Lecture 4
Images in ITK

Methods in Image Analysis
CMU Robotics Institute 16-725
U. Pitt Bioengineering 2630
Spring Term, 2006

Data storage in ITK

- ITK separates storage of data from the actions you can perform on data
- The DataObject class is the base class for the major “containers” into which you can place data

Data containers in ITK

- Images: N-d rectilinear grids of regularly sampled data
- Meshes: N-d collections of points linked together into cells (e.g. triangles)
- Meshes are outside the scope of this course, but please see section 4.3 of the ITK Software Guide for more information

What is an image?

- For our purposes, an image is an N-d rectilinear grid of data
- Images can contain *any* type of data, although scalars (e.g. grayscale) or vectors (e.g. RGB color) are most common
- We will deal mostly with scalars, but keep in mind that unusual images (e.g. linked-lists as pixels) are perfectly legal in ITK

Images are templated

```
itk::Image< TPixel, VImageDimension >
```

Pixel type

Dimensionality (value)

Examples:

```
itk::Image<double, 4>
itk::Image<unsigned char, 2>
```

An aside: smart pointers

- In C++ you typically allocate memory with new and deallocate it with delete
- Say I have a class called Cat:

```
Cat* pCat = new Cat;
pCat->Meow();
delete pCat;
```

Danger Will Robinson!

- Suppose you allocate memory in a function and forget to call delete prior to returning… the memory is still allocated, but you can’t get to it
- This is a *memory leak*
- Leaking doubles or chars can slowly consume memory, leaking 200 MB images will bring your computer to its knees

Smart pointers to the rescue

- Smart pointers get around this problem by allocating and deallocating memory for you
- You *do not* explicitly delete objects in ITK, this occurs automatically when they go out of scope
- Since you can't forget to delete objects, you can't leak memory

(ahem, well, you have to try harder at least)

Smart pointers, cont.

- This is often referred to as *garbage collection* - languages like Java have had it for a while, but it's fairly new to C++
- Keep in mind that this only applies to ITK objects - you can still leak arrays of floats/chars/widgets to your heart's content

Why are smart pointers smart?

- Smart pointers maintain a “reference count” of how many copies of the pointer exist
- If N_{ref} drops to 0, nobody is interested in the memory location and it’s safe to delete
- If $N_{ref} > 0$ the memory is *not* deleted, because someone still needs it

Scope

- It’s not just a mouthwash
- Refers to whether or not a variable still exists within a certain segment of the code
- Local vs. global
- Example: variables created within member functions typically have local scope, and “go away” when the function returns

Scope, cont.

- Observation: smart pointers are only deleted when they go out of scope (makes sense, right?)
- Problem: what if we want to “delete” a SP that has *not* gone out of scope; there are good reasons to do this, e.g. loops

Scope, cont.

- You can create local scope by using {}
- Instances of variables created within the {} will go out of scope when execution moves out of the {}
- Therefore… “temporary” smart pointers created within the {} will be deleted
- Keep this trick in mind, you may need it

A final caveat about scope

- Don’t obsess about it
- 99% of the time, smart pointers are smarter than you!
- 1% of the time you may need to haul out the previous trick

Images and regions

- ITK was designed to allow analysis of very large images, even images that far exceed the available RAM of a computer
- For this reason, ITK distinguishes between an entire image and the part which is actually resident in memory or requested by an algorithm

Image regions

- Algorithms only process a region of an image that sits inside the current buffer
- The BufferedRegion is the portion of image in physical memory
- The RequestedRegion is the portion of image to be processed
- The LargestPossibleRegion describes the entire dataset

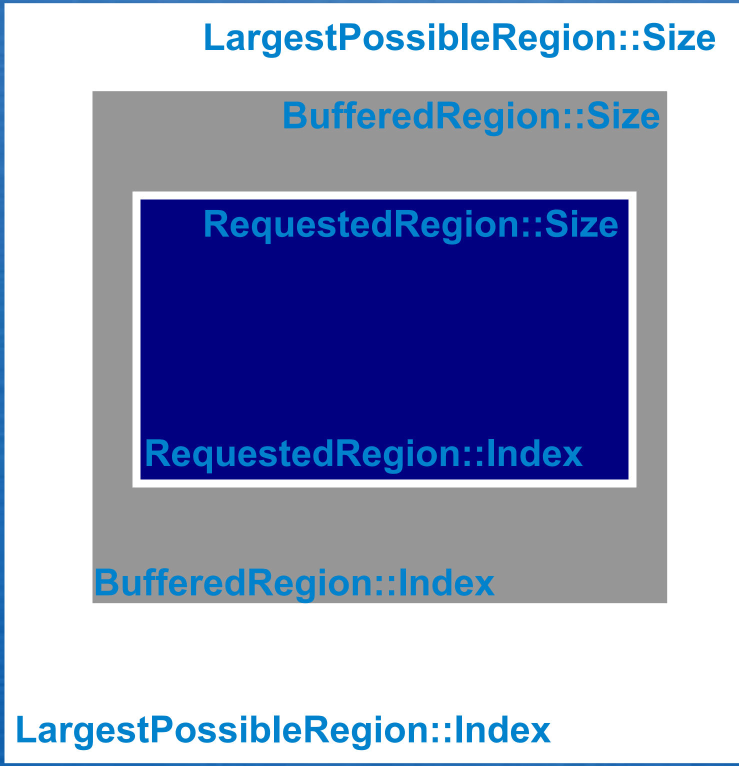


Image regions, cont.

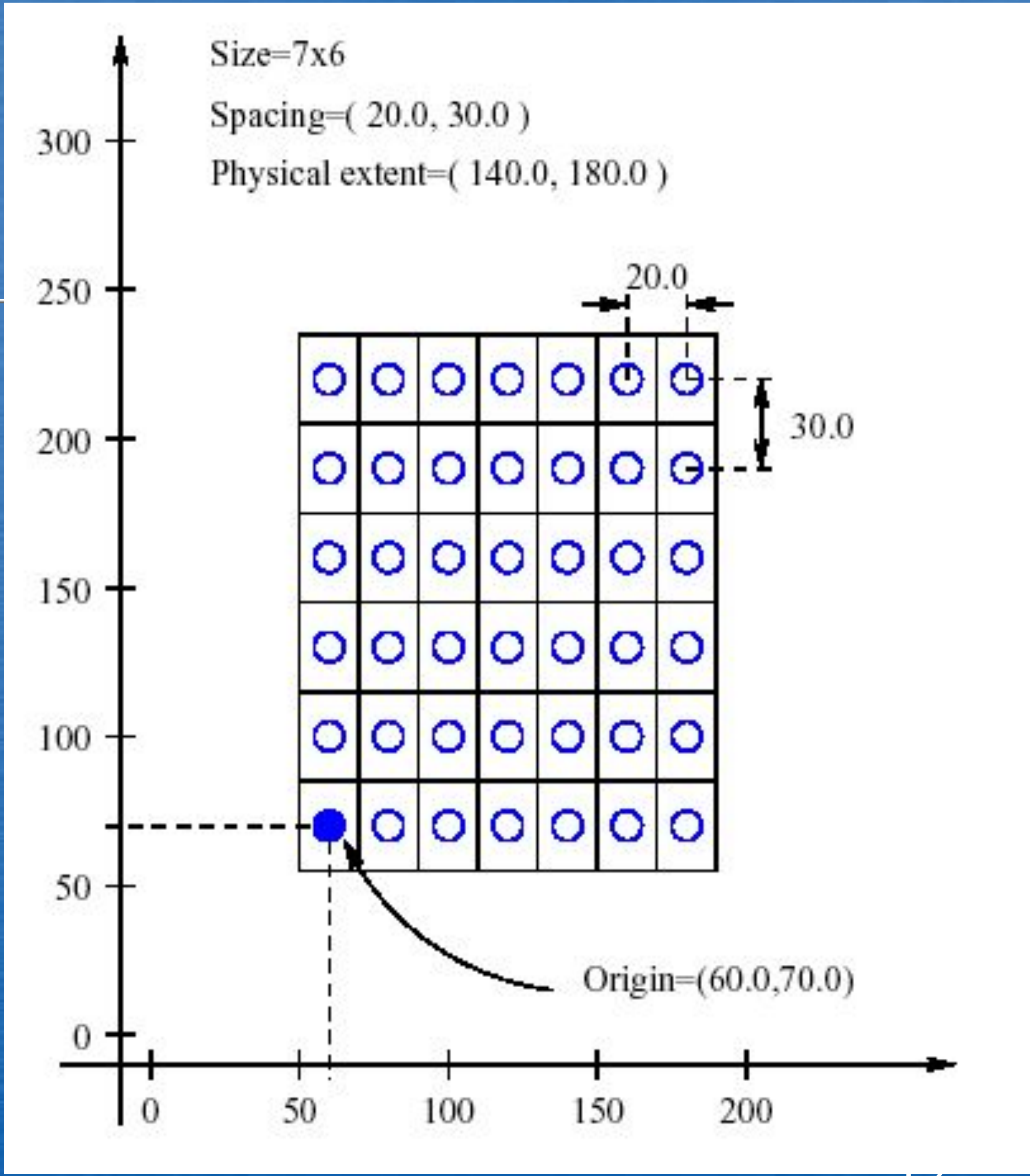
- It may be helpful for you to think of the LargestPossibleRegion as the “size” of the image
- When creating an image from scratch, you must specify sizes for all three regions - they *do not* have to be the same size
- Don’t get too concerned with regions just yet, we’ll look at them again with filters

Data space vs. "physical" space

- Data space is an N-d array with integer indices, indexed from 0 to (L_i - 1)
  - e.g. pixel (3,0,5) in 3D space

Length of side i

- Physical space relates to data space by defining the origin and spacing of the image


Size=7x6
Spacing=(20.0, 30.0)
Physical extent=(140.0, 180.0)
20.0
30.0
Origin=(60.0,70.0)
300
250
200
150
100
50
0
0
50
100
150
200

Creating an image: step-by-step

- Note: this example follows 4.1.1 from the ITK Software Guide, but differs in content - please be sure to read the guide as well
- This example is provided more as a demonstration than as a practical example - in the real world images are often/usually provided to you from an external source rather than being explicitly created

Declaring an image type

Recall the typename keyword… we first define an image type to save time later on:

```
typedef itk::Image< unsigned short, 3 >
  ImageType;
```

We can now use ImageType in place of the full class name, a nice convenience

A syntax note

It may surprise you to see something like the following:

(well, not if you were paying attention last week!)

```
ImageType::SizeType
```

Classes can have typedefs as members. In this case, SizeType is a public member of itk::Image. Remember that ImageType is itself a typedef, so we could express the above more verbosely as

```
itk::Image< unsigned short, 3 >::SizeType
```

Syntax note, cont.

- This illustrates one criticism of templates and typedefs - it's easy to invent something that looks like a new programming language!
- Remember that names ending in "Type" are types, not variables or class names
- Doxygen is your friend - you can find user-defined types under "Public Types"

Creating an image pointer

An image is created by invoking the New() operator from the corresponding image type and assigning the result to a SmartPointer.

```
ImageType::Pointer image = ImageType::New();
```

Pointer is typedef'd in itk::Image

Note the use of "big New"

A note about “big New”

- Many/most classes within ITK (indeed, all which derive from itk::Object) are created with the ::New() operator, rather than new

```
MyType::Pointer p = MyType::New();
```

- Remember that you should not try to call delete on objects created this way

When not to use ::New()

- "Small" classes, particularly ones that are intended to be accessed many (e.g. millions of) times will suffer a performance hit from smart pointers
- These objects can be created directly (on the stack) or using `new` (on the free store)

Setting up data space

The ITK Size class holds information about the size of image regions

SizeType is another typedef

```
ImageType::SizeType size;
size[0] = 200; // size along X
size[1] = 200; // size along Y
size[2] = 200; // size along Z
```

Setting up data space, cont.

Our image has to start somewhere - how about the origin?

```
ImageType::IndexType start;
start[0] = 0; // first index on X
start[1] = 0; // first index on Y
start[2] = 0; // first index on Z
```

Note that the index object *start* was not created with ::New()

Setting up data space, cont.

Now that we've defined a size and a starting location, we can build a region.

```
ImageType::RegionType region;
region.SetSize( size );
region.SetIndex( start );
```

region was also not created with ::New()

Allocating the image

Finally, we're ready to actually create the image. The SetRegions function sets all 3 regions to the same region and Allocate sets aside memory for the image.

```
image->SetRegions( region );
image->Allocate();
```

Dealing with physical space

- At this point we have an image of “pure” data; there is no relation to the real world
- Nearly all useful medical images are associated with physical coordinates of some form or another
- As mentioned before, ITK uses the concepts of origin and spacing to translate between physical and data space

Image spacing

We can specify spacing by calling the SetSpacing function in Image.

```
double spacing[ ImageType::ImageDimension ];
spacing[0] = 0.33; // spacing in mm along X
spacing[1] = 0.33; // spacing in mm along Y
spacing[2] = 1.20; // spacing in mm along Z
image->SetSpacing( spacing );
```

Image origin

Similarly, we can set the image origin

```
double origin[ImageType::ImageDimension];
origin[0] = 0.0; // coordinates of the
origin[1] = 0.0; // first pixel in N-D
origin[2] = 0.0;
image->SetOrigin( origin );
```

Origin/spacing units

- There are no inherent units in the physical coordinate system of an image - I.e. referring to them as mm's is arbitrary (but very common)
- Unless a specific algorithm states otherwise, ITK does not understand the difference between mm/inches/miles/etc.

Direct pixel access in ITK

- There are many ways to access pixels in ITK
- The simplest is to directly address a pixel by knowing either its:
  - Index in data space
  - Physical position, in physical space

Why *not* to directly access pixels

- Direct pixels access is simple conceptually, but involves a lot of extra computation (converting pixel indices into a memory pointer)
- There are much faster ways of performing sequential pixel access, through iterators

Accessing pixels in data space

- The Index object is used to access pixels in an image, in data space

```
ImageType::IndexType pixelIndex;
pixelIndex[0] = 27; // x position
pixelIndex[1] = 29; // y position
pixelIndex[2] = 37; // z position
```

Pixel access in data space

- To set a pixel:

```
ImageType::PixelType pixelValue = 149;
image->SetPixel(pixelIndex, pixelValue);
```

(the type of pixel stored in the image)

- And to get a pixel:

```
ImageType::PixelType value = image
  ->GetPixel( pixelIndex );
```

Why the runaround with PixelType?

- It might not be obvious why we refer to `ImageType::PixelType` rather than (in this example) just say `unsigned short`
- In other words, what's wrong with…?

```
unsigned short value = image->GetPixel(
  pixelIndex );
```

PixelType, cont.

- Well… nothing's wrong in this example
- *But*, in the general case we don't always know or control the type of pixel stored in an image
- Referring to ImageType will allow the code to compile for any type that defines the = operator (float, int, char, etc.)

PixelType, cont.

That is, if you have a 3D image of doubles,

```
ImageType::PixelType value = image
  ->GetPixel( pixelIndex );
```

works fine, while

```
unsigned short value = image->GetPixel(
  pixelIndex );
```

will produce a compiler warning

Walking through an image - Part 1

If you've done image processing before, the following pseudocode should look familiar:

```
loop over rows
  loop over columns
   build index (row, column)
   GetPixel(index)
  end column loop
end row loop
```

Image traversal, cont.

- The loop technique is easy to understand but:
  - Is slow
  - Doesn’t scale to N-d
  - Is unnecessarily messy from a syntax point of view
- Next week we’ll learn a way around this

Accessing pixels in physical space

ITK uses the Point class to store the position of a point in N-d space; conveniently, this is the “standard” for many ITK classes

```
typedef itk::Point< double,
  ImageType::ImageDimension >
  PointType;
```

Defining a point

Hopefully this syntax is starting to look somewhat familiar…

```
PointType point;
point[0] = 1.45; // x coordinate
point[1] = 7.21; // y coordinate
point[2] = 9.28; // z coordinate
```

Why do we need a Point?

- The image class contains a number of convenience methods to convert between pixel indices and physical positions (as stored in the Point class)
- These methods take into account the origin and spacing of the image, and do bounds-checking as well (I.e., is the point even inside the image?)

TransformPhysicalPointToIndex

- This function takes as parameters a Point (that you want) and an Index (to store the result in) and returns true if the point is inside the image and false otherwise
- Assuming the conversion is successful, the Index contains the result of mapping the Point into data space

The transform in action

First, create the index:

```
ImageType::IndexType pixelIndex;
```

Next, run the transformation:

```
image->TransformPhysicalPointToIndex(
   point,pixelIndex );
```

Now we can access the pixel!

```
ImageType::PixelType pixelValue =
   image->GetPixel( pixelIndex );
```

Point and index transforms

2 methods deal with integer indices:

`TransformPhysicalPointToIndex`

`TransformIndexToPhysicalPoint`

And 2 deal with floating point indices (used to interpolate pixel values):

`TransformPhysicalPointToContinuousIndex`

`TransformContinuousIndexToPhysicalPoint`