Programming for Image Processing/Analysis and Visualization using The Visualization Toolkit

Week 6: Case Study 1
The Iterative Closest Point Algorithm

http://noodle.med.yale.edu/seminar/seminar.html

Xenios Papademetris
xenios.papademetris@yale.edu
TAC (ex CAB) N119 5-6148

Schedule – Part 2
1. Review of Part 1 and Course Overview
2. C++ Pointers/Classes, Object Oriented Programming
3. VTK – an object library
4. Adding new VTK Commands/Cmake
5. Image-to-image filters
6. Case Study I -- Iterative Closest Point surface matching
7. Case Study II – A Simple Segmentation Algorithm

Talk Outline

• A Review of last few lectures
• A review of the VTK Transformation Classes
• Key Pieces
  • vtkLandmarkTransform
  • vtkCellLocator
• The ICP Algorithm
• The VTK implementation of ICP

Variables and Arrays

• C++ has basic data types such as short, int, float, char, double etc.

• The statements
  
  int a;
  float b[10];
  double c[5][5];

  define a single integer a, an one-dimensional array of floats b[0] .. b[9] and a two-dimensional array of doubles c[0][0] .. c[4][4] (All array indices start at 0.)

• Both a, b and c are implicitly allocated and will be deallocated when the function containing them exits. Their sizes are fixed at compile time.

Dynamic Memory Allocation

• Dynamic allocation allows the creation of arrays whose size is determined at runtime (e.g. loading an image whose size can vary).

• It is one of the key to writing memory efficient programs.

• It is, arguable, also the biggest source of problems and crashes in most C++ code. Most of the problems are caused by:
  1. Accessing/Deleting arrays/objects before they are allocated and initialized.
  2. Accessing/Deleting arrays/objects after they have been already deleted.

Allocating/De-allocating Pointers

• Memory allocation is performed by the new command e.g.
  
  int* a = new int;

• a is NOT an integer. It is a pointer which contains a memory location. To access the value stored at the memory location pointed by a use the de-referencing operator * e.g.

  *a = 1
  //This sets the value of the integer whose location is stored in a to 1

• When a is no longer needed the memory can be released using the delete command e.g.

  delete a;
Allocating/De-allocating Arrays of Pointers

- As before allocation is performed by the `new` command e.g.
  
  ```
  int* a = new int[10]; // Allocate an array of 10 integers
  ```
  
- `a` is a pointer which contains the memory location of the first element of the array. To access the values stored at the memory locations use:
  
  ```
  a[0] = 1; a[3] = 2; etc.
  ```
  
- When `a` is no longer needed the memory can be released using the `delete []` command e.g.
  
  ```
  delete [] a;
  ```
  
- Do not use the `delete` operator to delete arrays, it causes lots of problems use `delete []` !!!

Class Design

Classes consists of:

1. Data members -- just like C structures
2. Methods -- special functions which are embedded in the class

Members and methods can be declared as public which makes them accessible from outside the class or protected/private which makes them inaccessible from outside the class methods.

VTK and Macros

- VTK uses certain macros to help with the parsing of the C++ header files and the creation of wrapper code for TCL/Java/Python.

  ```
  #define VTKMacro(member, type) 
  typedef type member;
  ```

  e.g. `vtkSetMacro(Length, float)` results in

  ```
  void SetLength(float f) { Length = f; }
  ```

  and `vtkGetMacro(Length, float)` results in

  ```
  float GetLength() { return Length; }
  ```

  `vtkSetObjectMacro(object, type)` and `vtkGetObjectMacro(object, type)` are used to access/modify pointer objects derived from `vtkObject`

The vtkDataObject Branch

VTK Pipeline

- **Sources**
- **Filters**
- **Mappers**
- **Props**
- **File Output**

Two key filter methods:

- `ExecuteInformation()`
- `Execute()`

Talk Outline

- A Review of last few lectures
- A review of the VTK Transformation Classes
- Key Pieces
  - `vtkLandmarkTransform`
  - `vtkCellLocator`
- The ICP algorithm
- The VTK implementation of ICP
  `vtkIterativeClosestPointTransform`
The **vtkAbstractTransform Branch**

- **vtkAbstractTransform**
  - Abstract parentclass of all VTK transformations
  - Defines interface (but not the implementation) for most common operations e.g.
    - TransformPoint()
    - Identity()
    - Inverse()
    - TransformDerivative()
    - TransformNormalAtPoint()
  - **vtkHomogeneousTransform** provides a generic interface for homogeneous transformations, i.e. transformations which can be represented by multiplying a 4x4 matrix with a homogeneous coordinate.

The **vtkHomogeneousTransform Branch**

- **vtkIterativeClosestPointTransform**
  - Derived from **vtkLinearTransform**
  - It computes an "optimal" 4x4 matrix which defines the transformation between two data sets (stored as **vtkDataSet**) by iteratively estimating the correspondence between the point sets

- From the VTK man page:
  "Match two surfaces using the iterative closest point (ICP) algorithm. The core of the algorithm is to match each vertex in one surface with the closest surface point on the other, then apply the transformation that modify one surface to best match the other (in a least square sense). This has to be iterated to get proper convergence of the surfaces.

  **Attention:**
  Use **vtkTransformPolyDataFilter** to apply the resulting ICP transform to your data. This class makes use of **vtkLandmarkTransform** internally to compute the best fit. Use the GetLandmarkTransform member to get a pointer to that transform and set its parameters. You might, for example, constrain the number of degrees of freedom of the solution (i.e. rigid body, similarity, etc.) by checking the **vtkLandmarkTransform** documentation for its SetMode member."

**Talk Outline**

- A Review of last few lectures
- A review of the VTK Transformation Classes
- Key Pieces
  - **vtkLandmarkTransform**
  - **vtkCellLocator**
- The ICP algorithm
- The VTK implementation of ICP

**vtkLandmarkTransform**

- Derived from **vtkLinearTransform**
- It computes an "optimal" 4x4 matrix which defines the transformation between two sets of point (stored as **vtkPoints**) which have the same number of points and are assumed to be in correspondence.

- From the VTK man page:
  "A **vtkLandmarkTransform** is defined by two sets of landmarks, the transform computed gives the best fit mapping one onto the other, in a least squares sense. The indices are taken to correspond, so point 1 in the first set will get mapped close to point 1 in the second set, etc. Call SetSourceLandmarks and SetTargetLandmarks to specify the two sets of landmarks, ensure they have the same number of points.

  **Warning:**
  Whenever you add, subtract, or set points you must call Modified() on the **vtkPoints** object, or the transformation might not update."
**vtkLandmarkTransform**

- Key Methods:
  - void SetSourceLandmarks (vtkPoints *points)
  - void SetTargetLandmarks (vtkPoints *points)
  - void SetModeToRigidBody ()
  - void SetModeToSimilarity ()
  - void SetModeToAffine ()

- From vtkHomogeneousTransform
  - vtkMatrix4x4 * GetMatrix()
  - void Update()

**vtkLocator Tree**

From the man page: "vtkLocator is an abstract base class for spatial search objects, or locators. The principle behind locators is that they divide 3-space into small pieces (or "buckets") that can be quickly found in response to queries like point location, line intersection, or object-object intersection."

**vtkCellLocator**

- From the man page: "vtkCellLocator is a spatial search object to quickly locate cells in 3D. vtkCellLocator uses a uniform-level octree subdivision, where each octant (an octant is also referred to as a bucket) carries an indication of whether it is empty or not, and each leaf octant carries a list of the cells inside of it. Typical operations are intersection with a line to return candidate cells, or intersection with another vtkCellLocator to return candidate cells."

- A cell is a triangle/voxel etc depending on the dataset.

- Key methods:
  - void BuildLocator ()
  - void FindClosestPoint (float x[3], float closestPoint[3], vtkIdType &cellId, int &subId, float &dist2)

**Talk Outline**

- A Review of last few lectures
- A review of the VTK Transformation Classes
- Key Pieces
  - vtkLandmarkTransform
  - vtkCellLocator
- The ICP algorithm
- The VTK implementation of ICP

**Algorithm**

1. Initialization
   1. Take two points sets A and B
   2. Initialize the transformation series with $T_0$ as the identity.

2. Iteration $k$:
   1. For each point in set $A$, find the closest point in $B$ to it $q_k$.
   2. Estimate the best transformation $T_{k}$ in a least squares sense that minimizes the distance between ($p_i$ and $q_k$).
   3. Compute the average distance $d$ between the set of ($p_i$ and $q_k$) if less than threshold stop.
   4. Let $p_i=q_k$.

3. Final transformation $T$ is the concatenation of all $T_k$ i.e. $T = T_n \circ T_{n-1} \circ \ldots \circ T_1 \circ T_0$
vtkIterativeClosestPointTransform.h --1

- Standard stuff, some default constants and specification of inheritance etc

#include "vtkLinearTransform.h"

#define VTK_ICP_MODE_RMS 0
#define VTK_ICP_MODE_AV 1

class vtkCellLocator;
class vtkLandmarkTransform;
class vtkDataSet;
class vtkIterativeClosestPointTransform : public vtkLinearTransform
{
public:
  static vtkIterativeClosestPointTransform *New();
  vtkTypeMacro(vtkIterativeClosestPointTransform,vtkLinearTransform);
  void PrintSelf(ostream &os, vtkIndent indent);

  void SetSource(vtkDataSet *source);
  void SetTarget(vtkDataSet *target);
  void SetLocator(vtkCellLocator *locator);

  void SetMaximumNumberOfIterations(int);
  void SetCheckMeanDistance(int);

  void SetMeanDistanceModeToRMS();
  void SetMeanDistanceModeToAbsoluteValue();

  void SetMaximumMeanDistance(float);

  void SetMaximumNumberOfLandmarks(int);

  void StartByMatchingCentroids(int);

  vtkLandmarkTransform *GetLandmarkTransform();

  void Inverse();
  vtkAbstractTransform *MakeTransform();

private:

  vtkDataSet *Source;
  vtkDataSet *Target;
  vtkCellLocator *Locator;
  int MaximumNumberOfIterations;
  int CheckMeanDistance;
  int MeanDistanceMode;
  float MaximumMeanDistance;
  int MaximumNumberOfLandmarks;
  int NumberOfIterations;

  float MeanDistance;

  vtkLandmarkTransform *LandmarkTransform;

};

#endif

vtkIterativeClosestPointTransform.cxx

- We will examine in detail the following methods
  - Constructor and Destructor
  - SetSource() and ReleaseSource() which are examples of obtaining external VTK objects and modifying the number of references to them to ensure that
  1. The Source object does not get deleted accidentally (SetSource increases its reference number by 1)
  2. The Source object does not stay in memory when it is no longer used (ReleaseSource decreases its reference number by 1)
  3. Appropriate reference modification when the source object is changed!

  The pairs SetTaget()/ GetTarget() SetLocator / ReleaseLocator() are very similar to the SetSource()/ ReleaseSource()

- CreateDefaultLocator() for creating the Cell Locator if no external one is specified!
- InternalDeepCopy() for copying the object

And finally:
- InternalUpdate() where the registration is computed.
vtkIterativeClosestPointTransform.cxx – 1
Constructor and Destructor

The constructor sets all the default values.
```cpp
vtkIterativeClosestPointTransform::vtkIterativeClosestPointTransform() : vtkLinearTransform()
{
    this->Source = NULL;
    this->Target = NULL;
    this->Locator = NULL;
    this->LandmarkTransform = vtkLandmarkTransform::New();
    this->MaximumNumberOfIterations = 50;
    this->CheckMeanDistance = 0;
    this->MaximumMeanDistance = 0.01;
    this->MaximumNumberOfLandmarks = 200;
    this->StartByMatchingCentroids = 0;
    this->NumberOfIterations = 0;
    this->MeanDistance = 0.0;
}
```

The destructor releases all external inputs (Source, Target, Locator) and deletes the landmark transform object which was allocated in the constructor.
```cpp
vtkIterativeClosestPointTransform::~vtkIterativeClosestPointTransform()
{
    ReleaseSource();
    ReleaseTarget();
    ReleaseLocator();
    this->LandmarkTransform->Delete();
}
```

vtkIterativeClosestPointTransform.cxx – 2
SetSource() / ReleaseSource()
```cpp
void vtkIterativeClosestPointTransform::SetSource(vtkDataSet* source)
{
    // Case 1 – the current source is the same as the one the method is trying to set: do nothing
    if (this->Source == source)
    return;

    // Otherwise first let go of the current source object (if it is not NULL) before taking a new source
    if (this->Source)
        this->ReleaseSource();

    // If the input source is not NULL register it first and then set this->Source=source
    if (source)
        source->Register(this);
    this->Source = source;
}
```

```cpp
void vtkIterativeClosestPointTransform::ReleaseSource(void)
{
    if (this->Source)
    {
        this->Source->UnRegister(this);
        this->Source = NULL;
    }
}
```

vtkIterativeClosestPointTransform.cxx – 3
CreateDefaultLocator() & CreateInternalCopy()
```cpp
void vtkIterativeClosestPointTransform::CreateDefaultLocator()
{
    if (this->Locator)
    {
        this->ReleaseLocator();
    }
    this->Locator = vtkCellLocator::New();
}
```

```cpp
void vtkIterativeClosestPointTransform::InternalDeepCopy(vtkAbstractTransform* transform)
{
    vtkIterativeClosestPointTransform* t = (vtkIterativeClosestPointTransform*)transform;
    this->SetSource(t->GetSource());
    this->SetTarget(t->GetTarget());
    this->SetLocator(t->GetLocator());
    this->SetMaximumNumberOfIterations(t->GetMaximumNumberOfIterations());
    this->SetCheckMeanDistance(t->GetCheckMeanDistance());
    this->SetMeanDistanceMode(t->GetMeanDistanceMode());
    this->SetMaximumMeanDistance(t->GetMaximumMeanDistance());
    this->SetMaximumNumberOfLandmarks(t->GetMaximumNumberOfLandmarks());
    this->Modified();
}
```

vtkIterativeClosestPointTransform.cxx – 4
InternalUpdate()
```cpp
void vtkIterativeClosestPointTransform::InternalUpdate()
{
    // Check source, target
    if (this->Source == NULL || !this->Source->GetNumberOfPoints())
    {
        vtkErrorMacro("Can't execute with NULL or empty input");
        return;
    }

    if (this->Target == NULL || !this->Target->GetNumberOfPoints())
    {
        vtkErrorMacro("Can't execute with NULL or empty target");
        return;
    }

    // Create locator
    this->CreateDefaultLocator();
    this->Locator->SetDataSet(this->Target);
    this->Locator->SetNumberOfCellsPerBucket(1);
    this->Locator->BuildLocator();

    // Create two sets of points to handle iteration step=sampling rate
    int step = 1;
    if (this->Source->GetNumberOfPoints() > this->MaximumNumberOfLandmarks)
    {
        step = this->Source->GetNumberOfPoints() / this->MaximumNumberOfLandmarks;
    }

    int nb_points = this->Source->GetNumberOfPoints() / step;
    vtkPoints* points1 = vtkPoints::New();
    points1->SetNumberOfPoints(nb_points);
    vtkPoints* closestp = vtkPoints::New();
    closestp->SetNumberOfPoints(nb_points);
    vtkPoints* points2 = vtkPoints::New();
    points2->SetNumberOfPoints(nb_points);

    // Fill with initial positions (sample dataset using step)
    vtkTransform* accumulate = vtkTransform::New();
    accumulate->PostMultiply();

    // Allocate some points.
    vtkTypeId rb_points = this->Source->GetNumberOfPoints();
    step = this->Source->GetNumberOfPoints();
    for (int i = 0; i < rb_points; i++)
    {
        points1->SetPoint(i, closestp->GetPoint(i * step));
    }
    for (int i = 0; i < rb_points; i++)
    {
        points2->SetPoint(i, closestp->GetPoint(i * step));
    }
}```
Algorithm

1. Initialization
   1. Take two point sets A and B
   2. Initialize the transformation series with $T_0$ as the identity.

2. Iteration $k$
   1. For each point in set A, find the closest point in B to it $q_i$
   2. Estimate the best transformation $T_k$ in a least squares sense that minimizes the distance between $(p_i, q_i)$
   3. Compute the average distance $d$ between the set of $(p_i, q_i)$ If less than threshold stop.
   4. Let $p_i = q_i$

3. Final transformation $T$ is the concatenation of all $T_k$ i.e. $T = T_k T_{k-1} \cdots T_0$

---

Next Week

- Final Case Study:
  - Choices
    1. Image-intensity based registration
    2. Segmentation using snakes
    3. Levelsets? (Marcel)
  - The idea would be to show how the design process will work rather than focusing too much on the underlying code
  - Highlight key VTK tools for making the implementation of these easier