



Innovative Clinical Collaboration, Not Software
www.3DnetMedical.com

How to build a pipeline.

Rendering

Biotronics 3D
Analyze • Collaborate • Discover

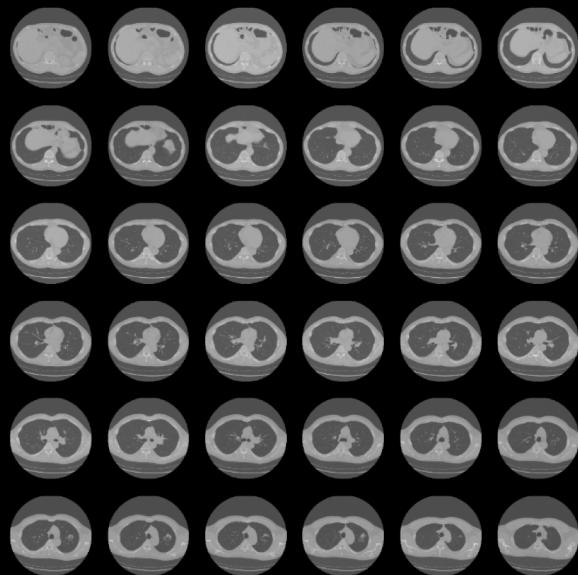
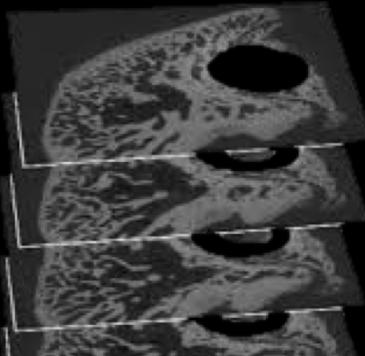
Understanding the requirements

- Fast efficient CPU rendering of CT (Computed Tomography), MR(I) (Magnetic Resonance Imaging).
- Support multiple volume rendering techniques. Windowing, MPR, CPR...,MIP, AvIP MinIP...Colour mapping.
- Volumetric unsigned 16bit data -> 8bit RGB bitmap buffer.
- Diagnostic quality.
- Isometric projection.

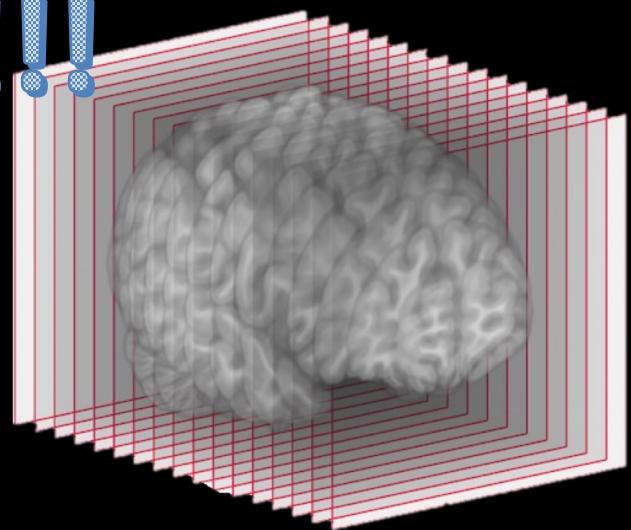
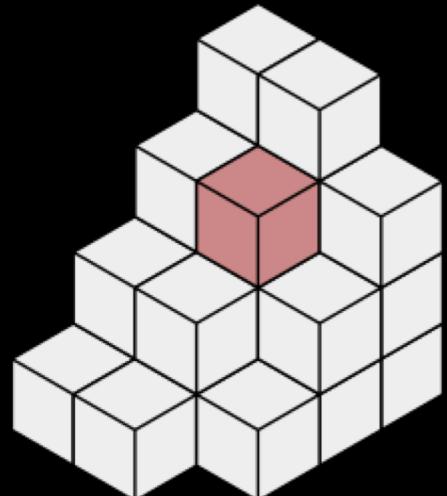
1. Understanding the requirements

- Volume Rendering.

Volumes formed of 2D slices, with spatial information.



TODO!!!!!!



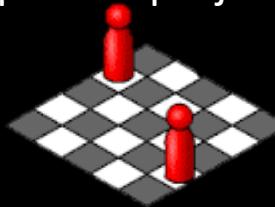
1. Understanding the requirements

- Isometric Projection

1. Perspective vs Isometric.
2. Preservation of distances is key for diagnostic quality.



Perspective projection



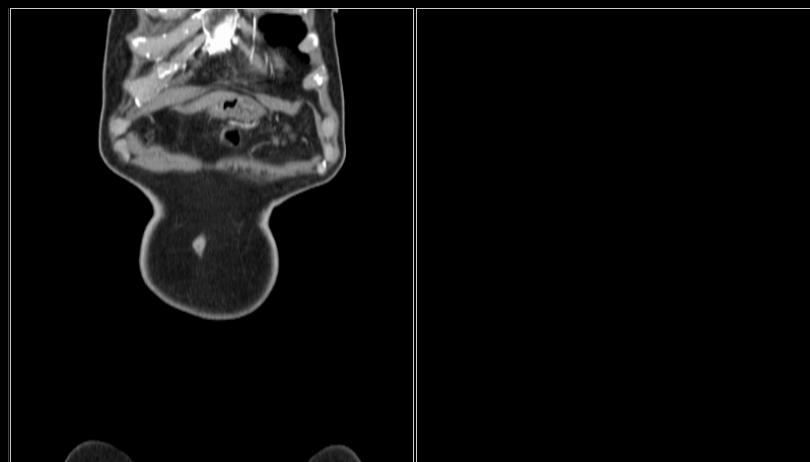
Isometric projection

TODO!!!!!!

1. Understanding the requirements

- Average Intensity Projection (AvIP)

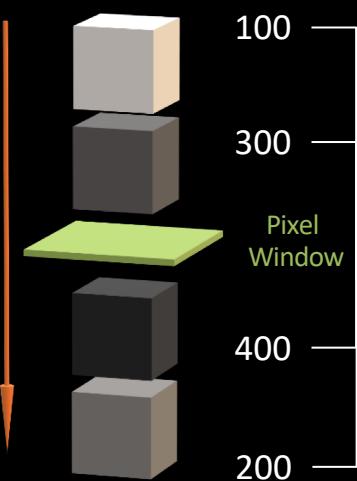
Tracing the ray through the pixel of interest, of n mm depth, take the average attenuation.



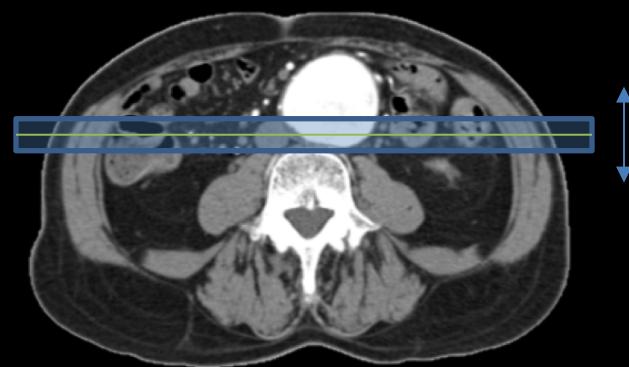
1mm AvIP

10mm AvIP

Pixel
Ray



$$\frac{1}{n} \sum_{i=1}^n a_i = \frac{1}{n} (a_1 + a_2 + \dots + a_n)$$
$$\frac{100 + 300 + 400 + 200}{4} = 250$$

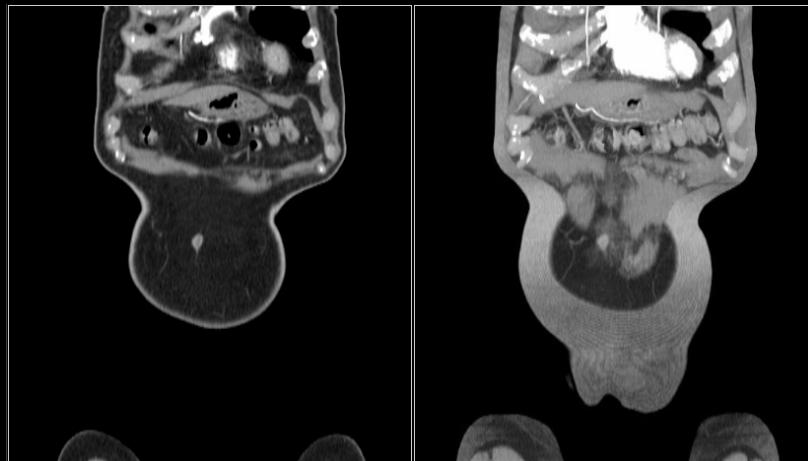


10mm

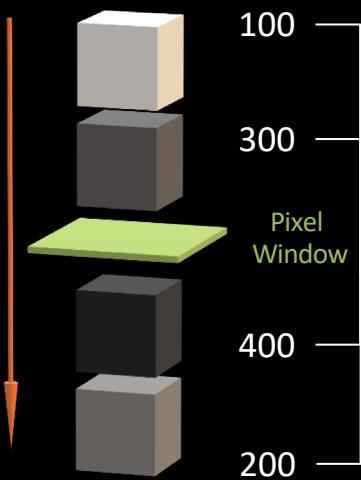
1. Understanding the requirements

- Maximum Intensity Projection (MIP)

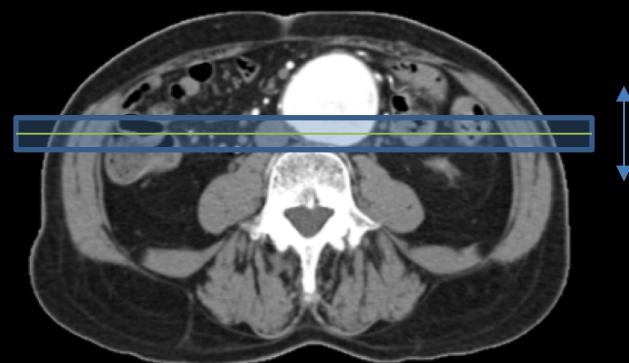
Tracing the ray through the pixel of interest, of n mm depth, take the largest attenuation.



Pixel
Ray



1mm MIP 10mm MIP

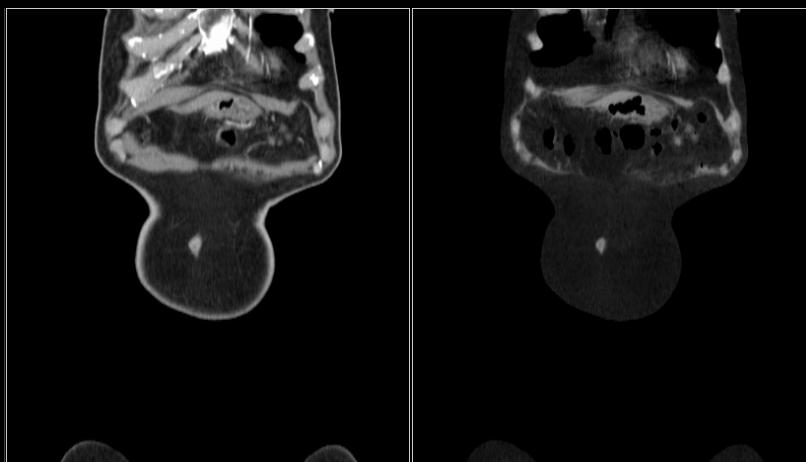


10mm

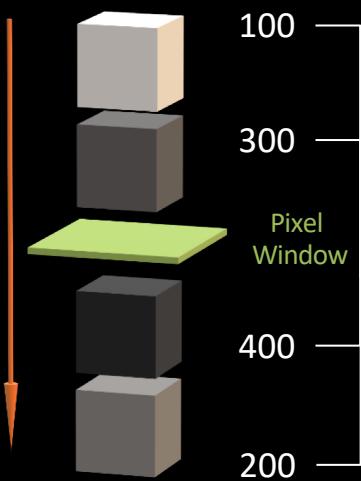
1. Understanding the requirements

- Minimum Intensity Projection (MinIP)

Tracing the ray through the pixel of interest, of n mm depth, take the lowest attenuation.



Pixel
Ray

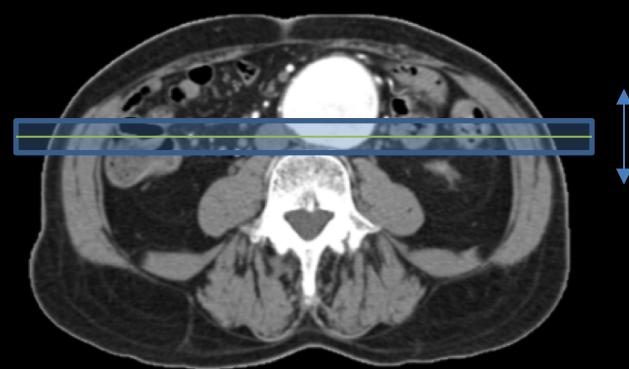


$$\min\{a_1, \dots, a_k\}$$

$$\min(100, 300, 400, 200) = 100$$

1mm MIP

10mm MIP



10mm

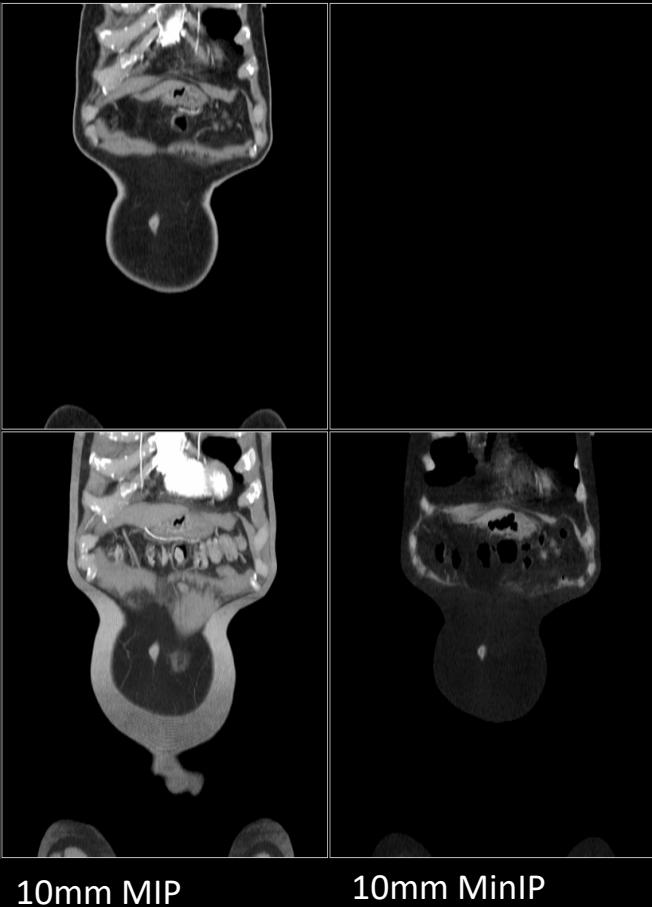
1. Understanding the requirements

- AvIP – Isodense structures are better represented. (Internal structures of a solid organs, Walls of hollow structures)
- MIP – Hyperdense structures are better represented. (Vessels with contrast, dense bone)
- MinIP – **Hypodense** structures are better represented. (Airways, Sinuses, Fat, Water)

Side by Side Comparison

1mm

10mm AvIP



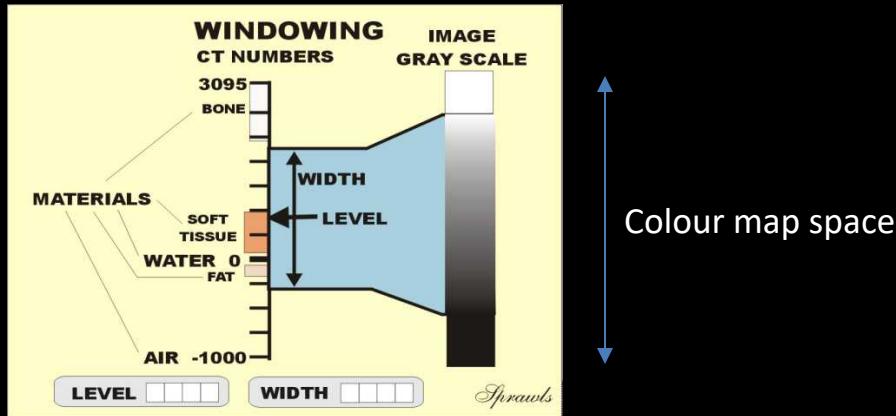
10mm MIP

10mm MinIP

1. Understanding the requirements

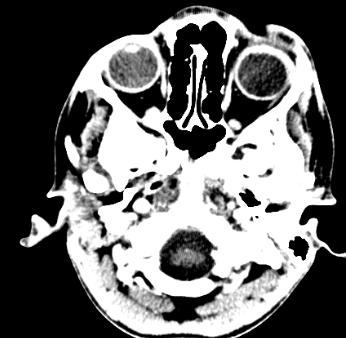
• Windowing / Window Level

Windowing (also known as **grey-level mapping**, **contrast stretching**, **histogram modification** or **contrast enhancement**) is the process in which the greyscale components are transformed; doing this will change the representation to highlight particular structures. Also, this allows us to convert our data our final colourspace.

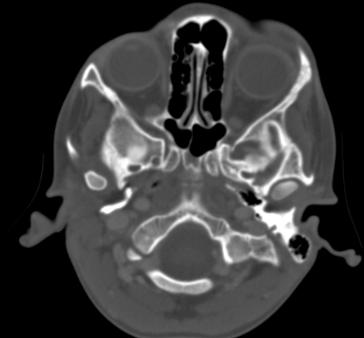


Colour map space

W:80 L:40 (Brain)



W:1800 L:400 (Bone)

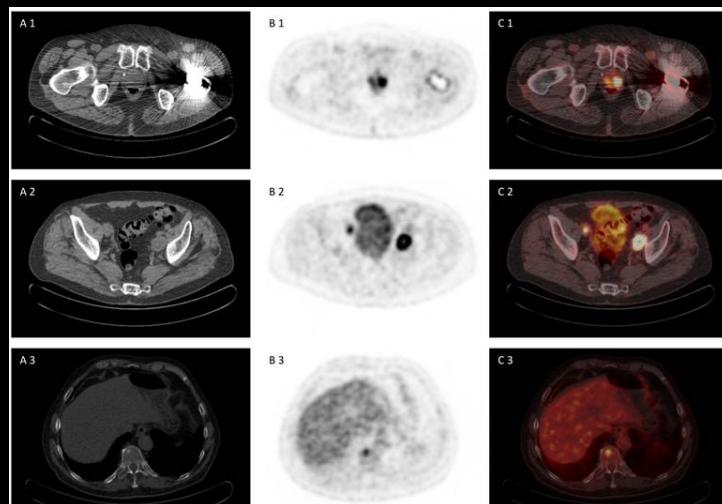
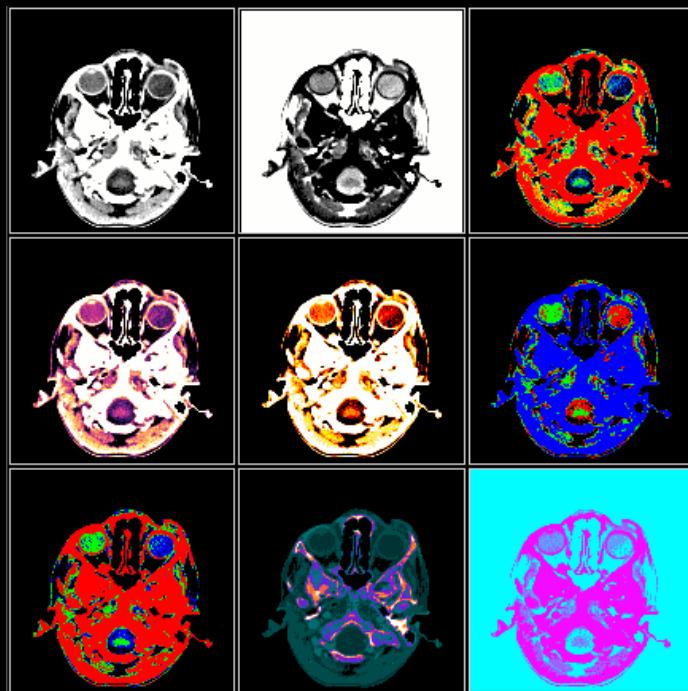
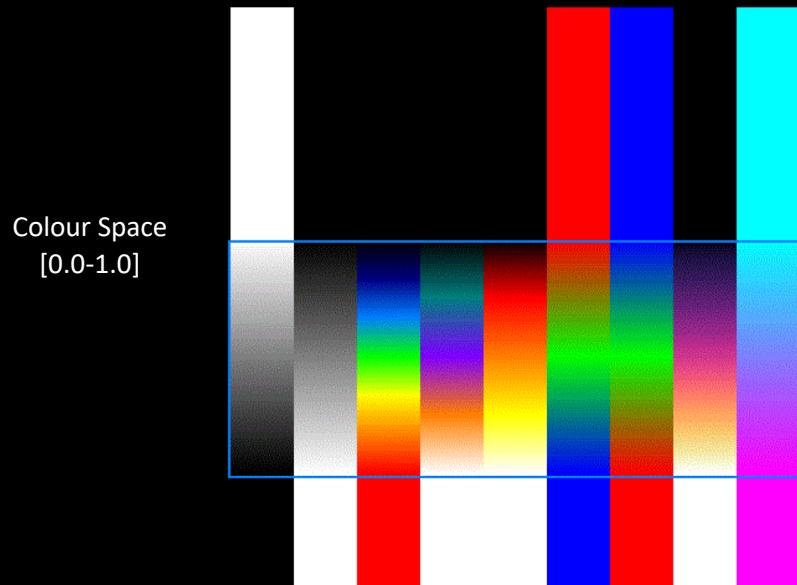


1. Understanding the requirements

• Colour Mapping

Provides additional clarity for windows of interest in the data.

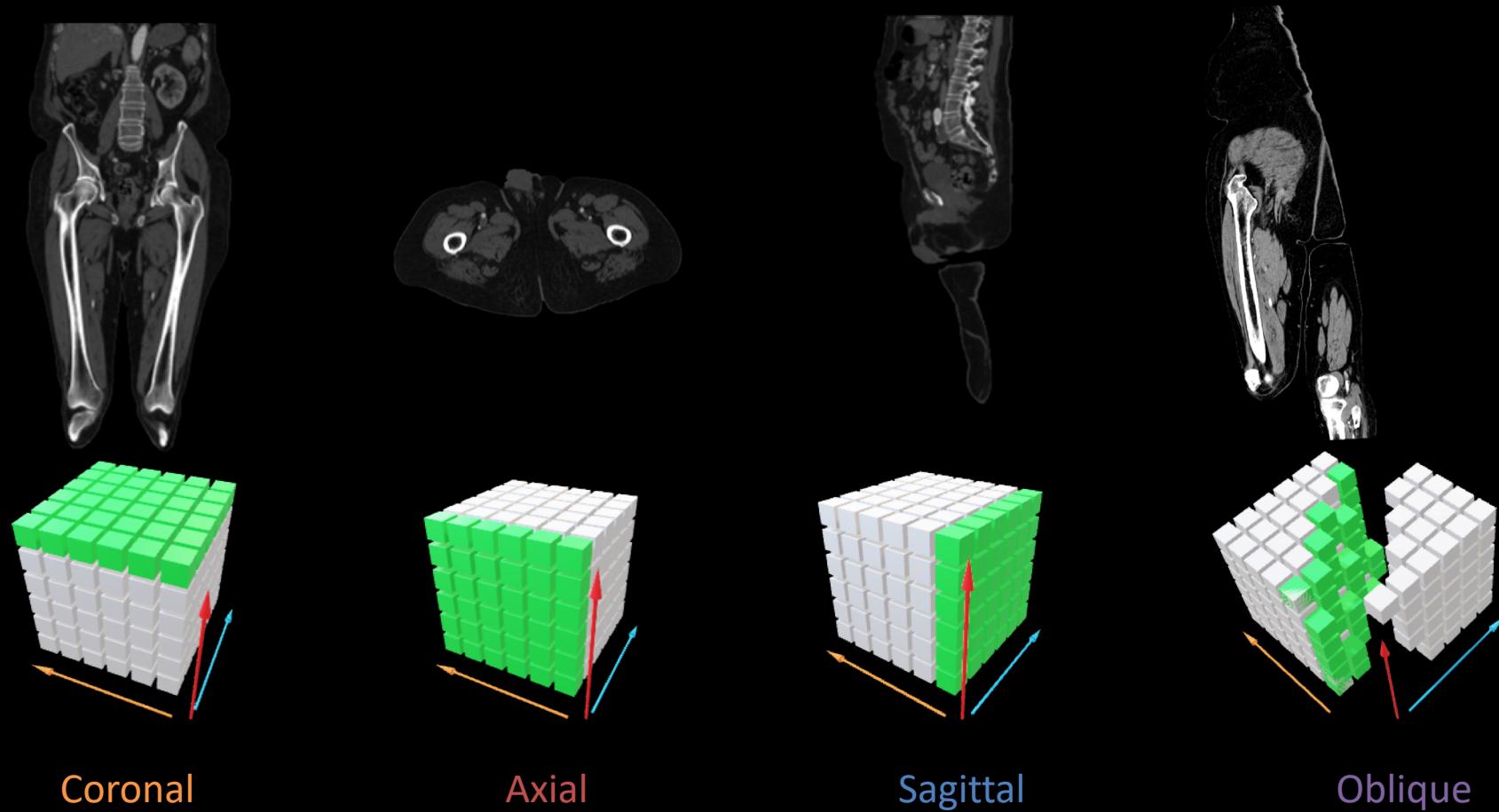
Additionally useful for PET/CT fusion, generating heat maps.



1. Understanding the requirements

- **Multiplanar Reconstruction (MPR)**

Multiplanar reformation (MPR) is the process of projecting a cross-section of a volume onto a flat plane or slab to produce images that are no longer aligned to the acquisition plane.



1. Understanding the requirements

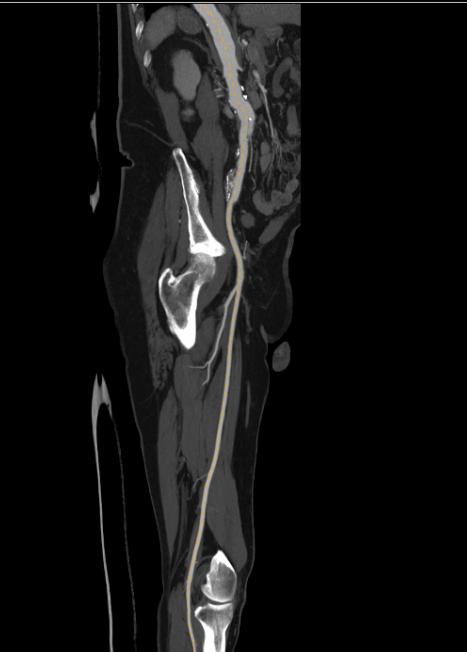
- Curved planar Reconstruction (CPR)

Curved planar reformation is a type of MPR accomplished by aligning the long axis of the imaging plane with a specific curved structure, such as a blood vessel, rather than with an arbitrary imaging plane.

Infinite MIP (Bone Removed)

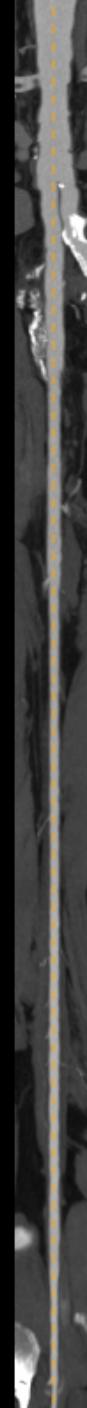
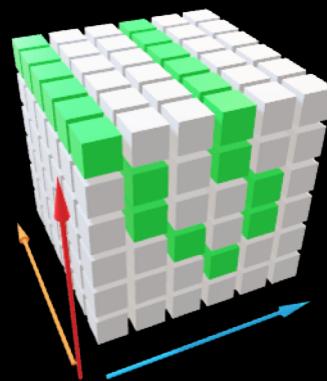


Stretched CPR



Straightened
CPR

Hyper Plane



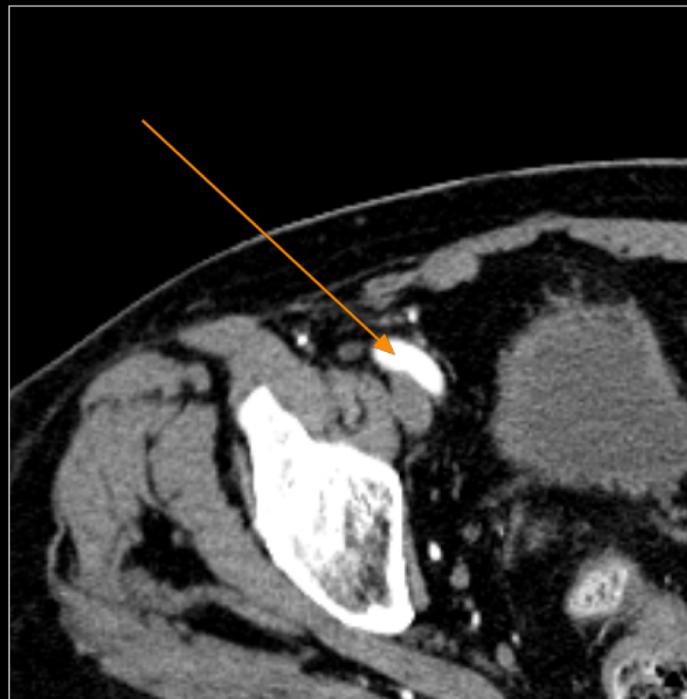
1. Understanding the requirements

Advantages of MPR Reconstruction

1. It enhances viewing of pathology.
2. It equips radiologists to deal with the large data sets that are available with the new multi-detector CT scanners and to more easily compare current and previous exams.
3. It improves service to referring physicians, since selected 3-D images can be attached to the radiology report. These images illustrate the diagnosis and may even be shown to patients while discussing the condition and recommended treatment.



Original Slice



Oblique Planes



2. Prototype MPR

```
10  class MPRGenerator {
11  public:
12      static void Run(
13          const float windowInv, const float windowLower,
14          const Vol3<ushort>& volume,
15          RGBA<byte>* imageBuffer, const Vec3i& imageDIM,
16          const Trans3f& imageToData) {
17              std::vector<ushort> rowBlock(imageDIM[0]);
18              const AABBoxf volumeBox(Vec3f::Zero(), (Vec3f)(volume.Dim() - Vec3i::One()));
19              const AABBoxf volumeBoxHalf(-Vec3f::Half(), (Vec3f)(volume.Dim() - Vec3i::One()) + Vec3f::Half());
20
21              // Iterate down the image.(Y)
22              for(int deltaY = 0; deltaY < imageDIM[1]; ++deltaY) {
23
24                  // Reset the row.
25                  for (auto begin = rowBlock.begin(), end = rowBlock.end(); begin != end; ++begin) {
26                      *begin = 0;
27                  }
28
29                  // Iterate through the image plane.(Z)
30                  for(int deltaZ = 0; deltaZ < imageDIM[2]; ++deltaZ) {
31
32                      // Iterate across the row.(X)
33                      for(int deltaX = 0; deltaX < imageDIM[0]; ++deltaX) {
34
35                          // Compute the voxel position.
36                          // Remember we align the image plane through the center of the slab.
37                          const Vec3i imagePos(deltaX, deltaY, deltaZ - imageDIM[2] / 2);
38                          const Vec3f voxelPos = imageToData * (Vec3f)imagePos;
39                          // We must be within 0.5 of an edge.
40                          if(AABBoxf::BoxContains(volumeBoxHalf, voxelPos)) {
41                              // We must clamp (to fix the 0.5 edges).
42                              const Vec3i voxelPosClamped = (Vec3i)volumeBox.Clamp(voxelPos);
43                              // Sample the value.
44                              const ushort dataVal = *(volume.Data() + (size_t)voxelPosClamped[0] +
45                              (size_t)volume.Dim()[0] * voxelPosClamped[1] +
46                              (size_t)volume.Dim()[0] * volume.Dim()[1] * voxelPosClamped[2]);
47                              rowBlock[deltaX] = std::max(rowBlock[deltaX], dataVal);
48                          }
49                      }
50                  }
51                  // Convert and copy the row block into the image buffer.
52                  for (auto begin = rowBlock.begin(), end = rowBlock.end(); begin != end; ++begin, ++imageBuffer) {
53                      const byte grayVal = std::max(0.f, std::min(255.f, ((float)(*begin) - windowLower) * windowInv));
54                      *imageBuffer = RGBA<byte>(grayVal, grayVal, grayVal, 255.f);
55                  }
56              }
57          }
58      };
}
```

2. Prototype MPR (Template Constant)

MPRGenerator::Run<ProjectionMode::MIP>

```
10 enum ProjectionMode {
11     MIP,
12     MINIP,
13     AVIP
14 };
15
16 class MPRGenerator {
17 public:
18     template<ProjectionMode projectionMode>
19     static void Run(
20         const float windowInv, const float windowLower,
21         const Vol3<ushort>& volume,
22         RGBA<byte>* imageBuffer, const Vec3i& imageDIM,
23         const Trans3f& imageToData) {
24         std::vector<ushort> rowBlock(imageDIM[0]);
25         const AABBx volumeBox(Vec3f::Zero(), (Vec3f)(volume.Dim() - Vec3i::One()));
26         const AABBx volumeBoxHalf(-Vec3f::Half(), (Vec3f)(volume.Dim() - Vec3i::One()) + Vec3f::Half());
27
28         // Iterate down the image.(Y)
29         for(int deltaY = 0; deltaY < imageDIM[1]; ++deltaY) {
30
31             // Reset the row.
32             for (auto begin = rowBlock.begin(), end = rowBlock.end(); begin != end; ++begin) {
33                 switch (projectionMode) {
34                     case ProjectionMode::MINIP:
35                         *begin = 0xffff;
36                         break;
37                     case ProjectionMode::MIP:
38                     case ProjectionMode::AVIP:
39                         *begin = 0;
40                         break;
41                 }
42             }
43
44             // Iterate through the image plane.(Z)
45             for(int deltaZ = 0; deltaZ < imageDIM[2]; ++deltaZ) {
46
47                 // Iterate across the row.(X)
48                 for(int deltaX = 0; deltaX < imageDIM[0]; ++deltaX) {
49
50                     // Compute the voxel position.
51                     // Remember we align the image plane through the center of the slab.
52                     const Vec3i imagePos(deltaX, deltaY, deltaZ - imageDIM[2] / 2);
53                     const Vec3f voxelPos = imageToData * (Vec3f)imagePos;
54
55                     // We must be within 0.5 of an edge.
56                     if(AABBx::BoxContains(volumeBoxHalf, voxelPos)) {
57                         // We must clamp (to fix the 0.5 edges).
58                         const Vec3i voxelPosClamped = (Vec3i)volumeBox.Clamp(voxelPos);
59
56                     // Sample the value.
57                     const ushort dataVal = *(volume.Data() + (size_t)voxelPosClamped[0] +
58                     (size_t)volume.Dim()[0] * voxelPosClamped[1] +
59                     (size_t)volume.Dim()[0] * volume.Dim()[1] * voxelPosClamped[2]);
60
61
62                     switch (projectionMode) {
63                         case ProjectionMode::MIP:
64                             rowBlock[deltaX] = std::max(rowBlock[deltaX], dataVal);
65                             break;
66                         case ProjectionMode::MINIP:
67                             rowBlock[deltaX] = std::min(rowBlock[deltaX], dataVal);
68                             break;
69                         case ProjectionMode::AVIP:
70                             rowBlock[deltaX] += rowBlock[deltaX];
71                             break;
72                     }
73
74                 }
75             }
76         }
77     }
78
79         if(projectionMode == ProjectionMode::AVIP) {
79             for (auto begin = rowBlock.begin(), end = rowBlock.end(); begin != end; ++begin) {
80                 *begin /= imageDIM[2];
81             }
82         }
83
84         // Convert and copy the row block into the image buffer.
85         for (auto begin = rowBlock.begin(), end = rowBlock.end(); begin != end; ++begin, ++imageBuffer) {
86             const byte grayVal = std::max(0.f, std::min(255.f, ((float)(*begin) - windowLower) * windowInv));
87             *imageBuffer = RGBA<byte>(grayVal, grayVal, grayVal, 255.f);
88         }
89     }
90 }
91 }
```

2. Prototype MPR (Polymorphic Templates)

```
template<class T>
class Accumulator {
public:
    inline virtual T Sample(const T v, const T r) const = 0;

    template<class Iter>
    inline void PreSample(Iter begin, const Iter end) const {
        for (; begin != end; ++begin) {
            *begin = (T)0;
        }
    }

    template<class Iter>
    inline void PostSample(Iter begin, const Iter end) const {}
};
```

```
4 #include "Accumulator.cpp"
5 #include <limits>
6
7 template<class T>
8 class MinIPAccumulator : public Accumulator<T> {
9 public:
10     inline T Sample(const T v, const T r) const {
11         return std::min(v, r);
12     }
13
14     template<class Iter>
15     inline void PreSample(Iter begin, const Iter end) const {
16         for (; begin != end; ++begin) {
17             *begin = std::numeric_limits<T>::max();
18         }
19     }
20 }
21 
```

```
4 #include "Accumulator.cpp"
5 #include <limits>
6
7 template<class T>
8 class MiPAccumulator : public Accumulator<T> {
9 public:
10     inline T Sample(const T v, const T r) const {
11         return std::max(v, r);
12     }
13
14     template<class Iter>
15     inline void PreSample(Iter begin, const Iter end) const {
16         for (; begin != end; ++begin) {
17             *begin = std::numeric_limits<T>::min();
18         }
19     }
20 }
21 
```

```
4 #include "Accumulator.cpp"
5 #include <limits>
6
7 template <class T>
8 class AvIPAccumulator : public Accumulator<T> {
9 public:
10     inline T Sample(const T v, const T r) const {
11         return v + r;
12     }
13
14     template<class Iter>
15     inline void PostSample(Iter begin, const Iter end) const {
16         const int len = end - begin;
17         for (; begin != end; ++begin) {
18             *begin /= len;
19         }
20     }
21 }
```

2. Prototype MPR (Polymorphic Templates)

MPRGenerator::Run<MIPIAccumulator>

```
12 template<class Accumulator>
13 static void Run(
14     const float windowInv, const float windowLower,
15     const Vol3ushort& volume,
16     RGBA<byte>* imageBuffer, const Vec3i& imageDIM,
17     const Trans3f& imageToData) {
18     std::vector<ushort> rowBlock(imageDIM[0]);
19     const AABBx0 volumeBox(Vec3f::Zero(), (Vec3f)(volume.Dim() - Vec3i::One()));
20     const AABBx0 volumeBoxHalf(-Vec3f::Half(), (Vec3f)(volume.Dim() - Vec3i::One()) + Vec3f::Half());
21     const Accumulator accumulator;
22
23     // Iterate down the image.(Y)
24     for(int deltaY = 0; deltaY < imageDIM[1]; ++deltaY) {
25
26         // Reset the row accumulators.
27         accumulator.PreSample(rowBlock.begin(), rowBlock.end());
28
29         // Iterate through the image plane.(Z)
30         for(int deltaZ = 0; deltaZ < imageDIM[2]; ++deltaZ) {
31
32             // Iterate across the row.(X)
33             for(int deltaX = 0; deltaX < imageDIM[0]; ++deltaX) {
34
35                 // Compute the voxel position.
36                 // Remember we align the image plane through the center of the slab.
37                 const Vec3i imagePos(deltaX, deltaY, deltaZ - imageDIM[2] / 2);
38                 const Vec3f voxelPos = imageToData * (Vec3f)imagePos;
39                 // We must be within 0.5 of an edge.
40                 if(AABBx0::BoxContains(volumeBoxHalf, voxelPos)) {
41                     // We must clamp (to fix the 0.5 edges).
42                     const Vec3i voxelPosClamped = (Vec3i)volumeBox.Clamp(voxelPos);
43                     // Sample the value.
44                     const ushort dataVal = *(volume.Data() + (size_t)voxelPosClamped[0] +
45                     (size_t)volume.Dim()[0] * voxelPosClamped[1] +
46                     (size_t)volume.Dim()[0] * volume.Dim()[1] * voxelPosClamped[2]);
47
48                     rowBlock[deltaX] = accumulator.Sample(rowBlock[deltaX], dataVal);
49                 }
50             }
51         }
52
53         // Complete the row accumulators.
54         accumulator.PostSample(rowBlock.begin(), rowBlock.end());
55
56         // Convert and copy the row block into the image buffer.
57         for (auto begin = rowBlock.begin(), end = rowBlock.end(); begin != end; ++begin, ++imageBuffer) {
58             const byte grayVal = std::max(0.f, std::min(255.f, ((float)(*begin) - windowLower) * windowInv));
59             *imageBuffer = RGBA<byte>(grayVal, grayVal, grayVal, 255.f);
60         }
61     }
62 }
```

2. Prototype MPR (Polymorphic Templates)

```
4     template<class Tin, class Tout>
5         class Converter {
6             public:
7                 Converter() {}
8                 virtual Tout Convert(const Tin v) const = 0;
9             };
10
```

```
4 #include "Converter.cpp"
5
6
7 template<class T>
8 class WLConverter : public Converter<T, float> {
9     float m_windowInv, m_lower_window_limit;
10
11 public:
12     WLConverter(const float window, const float level):
13         m_windowInv(1.0f / window),
14         m_lower_window_limit(level - window / 2) {}
15     float Convert(const T v) const {
16         return ((float)v - m_lower_window_limit) * m_windowInv;
17     }
18 };
19
20
21
22
23
24
25
26
27
4 #include "Converter.cpp"
5 #include "../RGBAlpha.cpp"
6
7 template<class T>
8 class ColourMapConverter : public Converter<float, RGBAlpha<T>> {
9     private:
10         const std::map<float, RGBAlpha<T>> m_colourLUT;
11         const float m_scalar;
12     public:
13         ColourMapConverter(const std::map<float, RGBAlpha<T>>& colourmap):
14             m_colourLUT(colourmap.begin(), colourmap.end()),
15             m_scalar(!colourmap.empty())
16             ? 1.0f / ((--colourmap.end()).first - (*colourmap.begin()).first)
17             : 1.0f {}
18         RGBAlpha<T> Convert(const float v) const {
19             const float vClamped = std::min(1.0f, std::max(0.0f, v));
20             const auto c0Iter = m_colourLUT.lower_bound(vClamped);
21             const auto c1Iter = m_colourLUT.upper_bound(vClamped);
22             const float t = (vClamped - (*c0Iter).first) * m_scalar;
23
24             return mix((*c0Iter).second, (*c1Iter).second, t);
25         }
26     };
27
4 #include "Converter.cpp"
5
6
7 template<class Tin, class Tout, class Cin, class Cout>
8 class InlineConverter : public Converter<Tin, Tout> {
9     private:
10     Cin m_in;
11     Cout m_out;
12     public:
13         InlineConverter(const Cin in, const Cout out):
14             m_in(in), m_out(out) {}
15         Tout Convert(const Tin v) const {
16             return m_out.Convert(m_in.Convert(v));
17         }
18
19
20
21
22
23
24
25
26
27
```

2. Prototype MPR (Polymorphic Templates)

```
7  template<class DataType>
8  class Vol3Sampler {
9  public:
10    virtual size_t GetOffsetAtPosition(const Vol3<DataType>& vol, const Vec3i& pos) const = 0;
11
12    virtual int GetXPositionAtOffset(const Vol3<DataType>& vol, const size_t offset) const = 0;
13
14    virtual int GetYPositionAtOffset(const Vol3<DataType>& vol, const size_t offset) const = 0;
15
16    virtual int GetZPositionAtOffset(const Vol3<DataType>& vol, const size_t offset) const = 0;
17
18    inline const DataType* GetPtrAtPosition(const Vol3<DataType>& vol, Vec3i pos) const {
19      assert(vol.Data() != nullptr);
20      return vol.Data() + GetOffsetAtPosition(vol, pos);
21    }
22
23    inline DataType* GetPtrAtPosition(Vol3<DataType>& vol, Vec3i pos) {
24      assert(vol.Data() != nullptr);
25      return vol.Data() + GetOffsetAtPosition(vol, pos);
26    }
27
28    inline const DataType& GetDataAtPosition(const Vol3<DataType>& vol, const Vec3i& pos) const {
29      return *GetPtrAtPosition(vol, pos);
30    }
31
32    inline DataType& GetDataAtPosition(Vol3<DataType>& vol, const Vec3i& pos) {
33      return *GetPtrAtPosition(vol, pos);
34    }
35
36    inline const DataType& GetDataAtOffset(const Vol3<DataType>& vol, const size_t offset) const {
37      assert(vol.Data() != nullptr);
38      return *(vol.Data() + offset);
39    }
40
41    inline DataType& GetDataAtOffset(Vol3<DataType>& vol, const size_t offset) {
42      assert((size_t)vol.Dim()[0] * vol.Dim()[1] * vol.Dim()[2] > offset);
43      return *(vol.Data() + offset);
44    }
45
46    inline size_t GetOffsetAtPtr(const Vol3<DataType>& vol, const DataType* const ptr) const {
47      assert((size_t)vol.Dim()[0] * vol.Dim()[1] * vol.Dim()[2] > (size_t)(ptr - vol.Data()));
48      return (size_t)(ptr - vol.Data());
49    }
50
51    inline Vec3i GetPositionAtOffset(const Vol3<DataType>& vol, const size_t offset) const {
52      return Vec3i(
53        GetXPositionAtOffset(vol, offset),
54        GetYPositionAtOffset(vol, offset),
55        GetZPositionAtOffset(vol, offset)
56      );
57    }
58
59    inline Vec3i GetPositionAtPtr(const Vol3<DataType>& vol, const DataType* const ptr) const {
60      return GetPositionAtOffset(GetOffsetAtPtr(vol, ptr));
61    }
62  };
63
64  #include "../GlobalDefs.h"
65  #include "Vol3Sampler.cpp"
66
67  template<class DataType>
68  class Vol3SamplerLinear : public Vol3Sampler<DataType> {
69  public:
70    inline size_t GetOffsetAtPosition(const Vol3<DataType>& vol, const Vec3i& pos) const {
71      assert(pos[0] < pos[0] >= 0);
72      assert(pos[0] < vol.Dim()[0]);
73      assert(pos[1] < pos[1] >= 0);
74      assert(pos[1] < vol.Dim()[1]);
75      assert(pos[2] < pos[2] >= 0);
76      assert(pos[2] < vol.Dim()[2]);
77
78      return (size_t)pos[0] +
79        (size_t)vol.Dim()[0] * pos[1] +
80        (size_t)vol.Dim()[0] * vol.Dim()[1] * pos[2];
81    }
82
83    inline int GetXPositionAtOffset(const Vol3<DataType>& vol, const size_t offset) const {
84      return (int)(offset % vol.Dim()[0]);
85    }
86
87    inline int GetYPositionAtOffset(const Vol3<DataType>& vol, const size_t offset) const {
88      return (int)(offset / vol.Dim()[0]) % vol.Dim()[1];
89    }
90
91    inline int GetZPositionAtOffset(const Vol3<DataType>& vol, const size_t offset) const {
92      return (int)(offset / vol.Dim()[0] / vol.Dim()[1]);
93    }
94  };
95
```

2. Prototype MPR (Polymorphic Templates)

```
3
4     template<class DataType, class VolType, class VolSampler>
5         class InterpKernal {
6             protected:
7                 const VolSampler m_sampler;
8             public:
9                 inline virtual DataType Sample(const VolType& volume, const Vec3f& position) const = 0;
10            };

```

```
4 #include "InterpKernal.cpp"
5
6 template<class DataType, class VolType, class VolSampler>
7 class NNInterpKernal : public InterpKernal<DataType, VolType, VolSampler> {
8 public:
9     inline DataType Sample(const VolType& volume, const Vec3f& position) const {
10         return this->m_sampler.GetDataAtPosition(volume, (Vec3i)position);
11     }
12 };
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
```

```
4 #include "InterpKernal.cpp"
5
6 template<class DataType, class VolType, class VolSampler>
7 class LIInterpKernal : public InterpKernal<DataType, VolType, VolSampler> {
8 public:
9     inline DataType Sample(const VolType& volume, const Vec3f& position) const {
10         const Vec3i p = (Vec3i)position;
11         const Vec3f t = position - (Vec3f)(Vec3i)position;
12
13         const float v0 = (float)this->m_sampler.GetDataAtPosition(volume, (Vec3i)position + Vec3i(0, 0, 0));
14         const float v1 = (float)this->m_sampler.GetDataAtPosition(volume, (Vec3i)position + Vec3i(1, 0, 0));
15         const float v2 = (float)this->m_sampler.GetDataAtPosition(volume, (Vec3i)position + Vec3i(0, 1, 0));
16         const float v3 = (float)this->m_sampler.GetDataAtPosition(volume, (Vec3i)position + Vec3i(1, 1, 0));
17         const float v4 = (float)this->m_sampler.GetDataAtPosition(volume, (Vec3i)position + Vec3i(0, 0, 1));
18         const float v5 = (float)this->m_sampler.GetDataAtPosition(volume, (Vec3i)position + Vec3i(1, 0, 1));
19         const float v6 = (float)this->m_sampler.GetDataAtPosition(volume, (Vec3i)position + Vec3i(0, 1, 1));
20         const float v7 = (float)this->m_sampler.GetDataAtPosition(volume, (Vec3i)position + Vec3i(1, 1, 1));
21
22         return (DataType)mix(
23             mix(mix(v0, v1, t[0]), mix(v2, v3, t[0]), t[1]),
24             mix(mix(v4, v5, t[0]), mix(v6, v7, t[0]), t[1]),
25             t[2]);
26
27     }
28 }
29
30 #endif
```

2. Prototype MPR (Polymorphic Templates)

```
6  class MPRGenerator {
7  public:
8    template<class VolType, class DataType, class PixelType, class Accumulator, class Converter, class InterpKernel>
9    static void Run(
10      const VolType& volume,
11      PixelType* imageBuffer,
12      const Vec3i& imageDIM,
13      const Trans3f& imageToData,
14      const Converter& converter) {
15        const Accumulator accumulator;
16        const InterpKernel interpKernel;
17        std::vector<DataType> rowBlock(imageDIM[0]);
18        const AABBBoxf volumeBox(Vec3f::Zero(), (Vec3f)(volume.Dim() - Vec3i::One()));
19        const AABBBoxf volumeBoxHalf(-Vec3f::Half(), (Vec3f)(volume.Dim() - Vec3i::One()) + Vec3f::Half());
20
21        // Iterate down the image.(Y)
22        for(int deltaY = 0; deltaY < imageDIM[1]; ++deltaY) {
23
24          // Reset the row accumulators.
25          accumulator.PreSample(rowBlock.begin(), rowBlock.end());
26
27          // Iterate through the image plane.(Z)
28          for(int deltaZ = 0; deltaZ < imageDIM[2]; ++deltaZ) {
29
30            // Iterate across the row.(X)
31            for(int deltaX = 0; deltaX < imageDIM[0]; ++deltaX) {
32
33              // Compute the voxel position.
34              // Remember we align the image plane through the center of the slab.
35              const Vec3i imagePos(deltaX, deltaY, deltaZ - imageDIM[2] / 2);
36              const Vec3f voxelPos = imageToData * (Vec3f)imagePos;
37              // We must be within 0.5 of an edge.
38              if(AABBBoxf::BoxContains(volumeBoxHalf, voxelPos)) {
39                // We must be within 0.5 of an edge.
40                const Vec3f voxelPosClamped = volumeBox.Clamp(voxelPos);
41                // Sample the value.
42                const DataType dataVal = interpKernel.Sample(volume, voxelPosClamped);
43                rowBlock[deltaX] = accumulator.Sample(rowBlock[deltaX], dataVal);
44              }
45            }
46          }
47
48          // Complete the row accumulators.
49          accumulator.PostSample(rowBlock.begin(), rowBlock.end());
50
51          // Convert and copy the row block into the image buffer.
52          for (auto begin = rowBlock.begin(), end = rowBlock.end(); begin != end; ++begin, ++imageBuffer) {
53            *imageBuffer = converter.Convert(*begin);
54          }
55        }
56      };
57    };
58  };
```

6. Optimization

- Blind optimization; don't do this.
- Function timing, understand the expensive code.
- Profiling, CPU cache hits/misses.

6. Optimization

Locality of reference

- At a minimum it takes one processor clock cycle to do each step. However, for steps 1 and 4 accessing main memory may take much longer than one cycle. Modern processors typically have a clock cycle of 0.5ns while accesses to main memory are 50ns or more. Thus, an access to main memory is very expensive, over 100 clock cycles. As result for performance critical sections the average times to fetch instructions and to access data from memory must be reduced.

