3dnet

Innovative Clinical Collaboration, Not Software

www.3DnetMedical.com

How to build a pipeline.

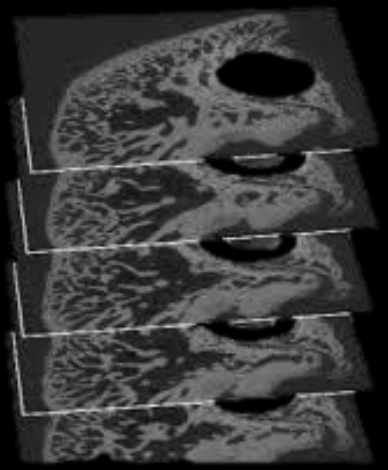
Biotronics 3D

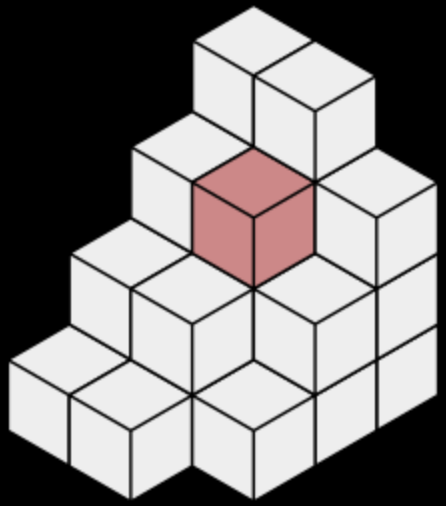
Analyze · Collaborate · Discover

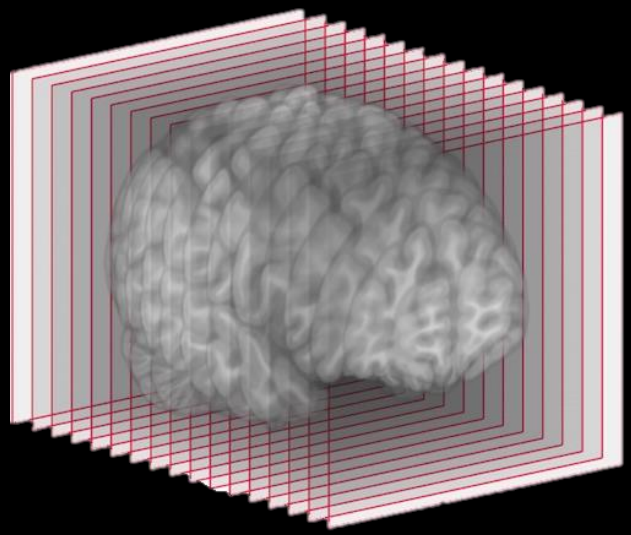
Volume Rendering

Scanner acquires 2D slices, with spatial information. Slices of constant distance and orientation are stacked to form our volume of interest. 3D pixels are referred to as “Voxels”.

Overall goal is to produce accurate, high quality, images of any cross-section from the volume.







1. Understanding the requirements

- Fast efficient rendering of CT (Computed Tomography), MR(I) (Magnetic Resonance Imaging) datasets.
- Support multiple volume rendering techniques. Windowing, MPR, CPR…,MIP, MinIP, AvIP…Colour mapping.
- Diagnostic quality (Interpolation, Isometric Projection)

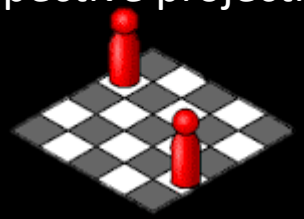
1. Understanding the requirements

- # Diagnosic quality.

Isometric Projection(3D):

1. Preservation of distances is crucial.
2. No warping of 3D space.
3. Measurements in real world. (Scaled)

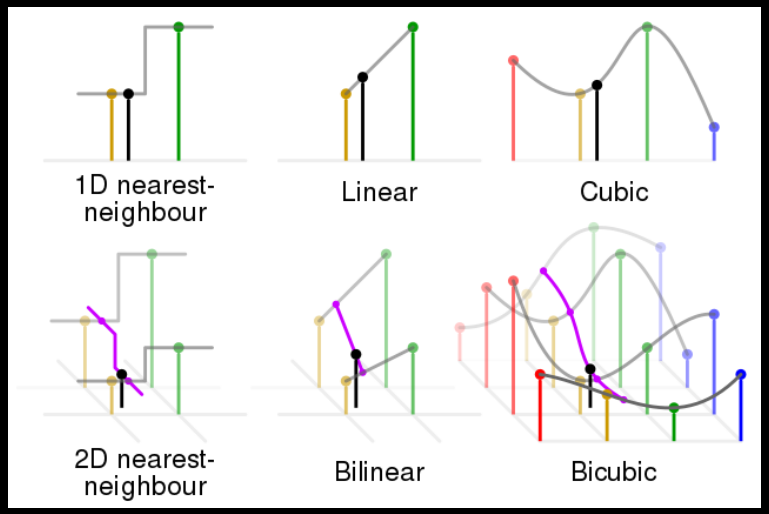
Perspective projection

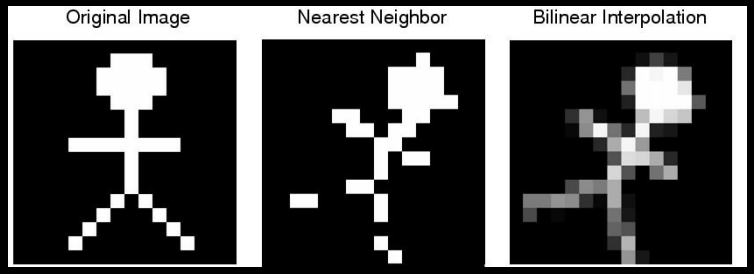


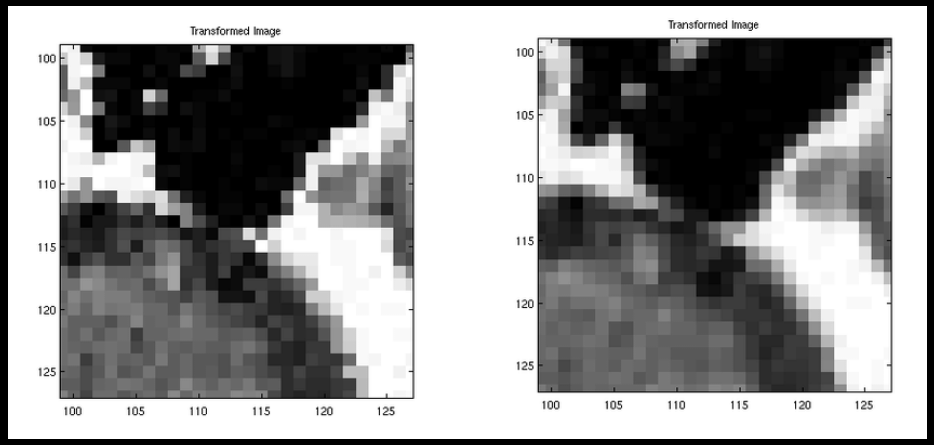
Isometric projection

- # Interpolation

1. Reduce aliasing effects.
2. Produce smooth images, form any orientation.

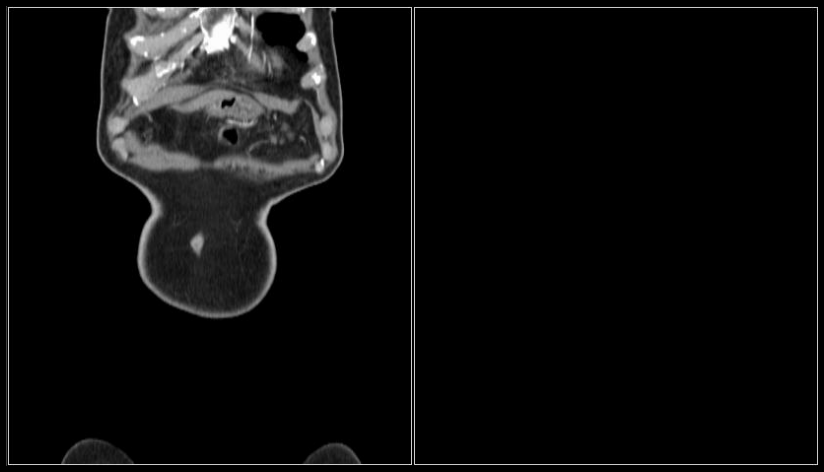






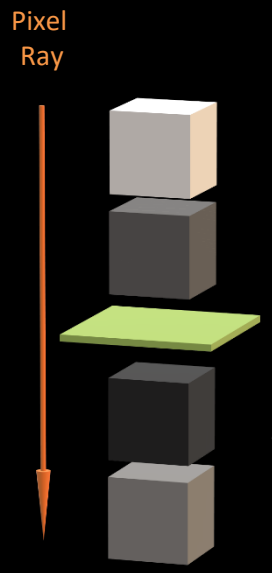
• Average Intensity Projection (AvIP)

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



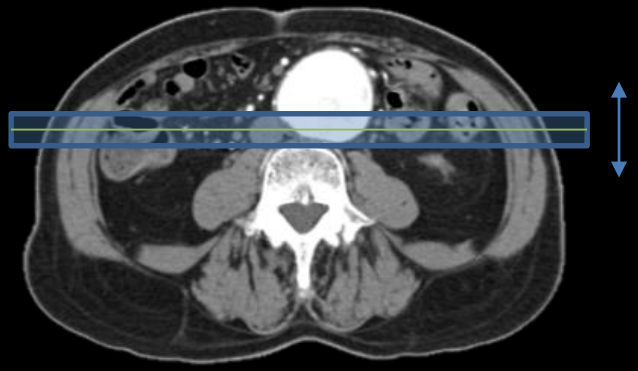
1mm AvIP

10mm AvIP



$$\frac{1}{n}\sum_{i=1}^{n} a_i = \frac{1}{n}(a_1 + a_2 + \cdots + a_n)$$

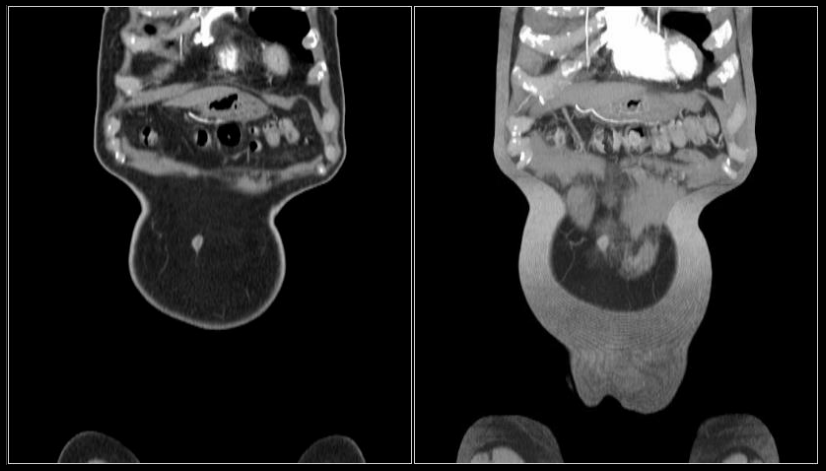
$$\frac{100 + 300 + 400 + 200}{4} = 250$$



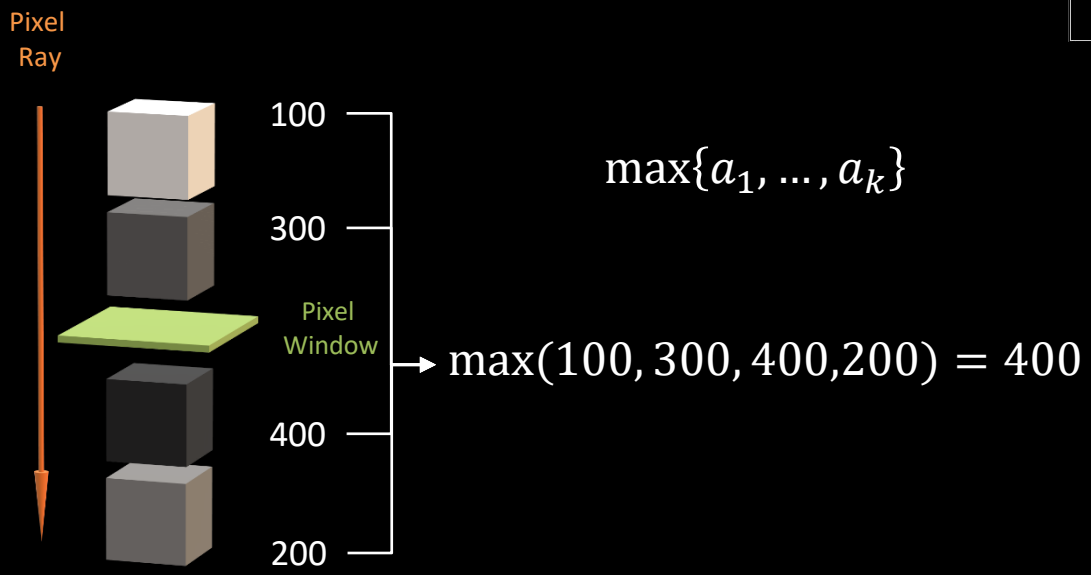
10mm

• Maximum Intensity Projection (MIP)

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

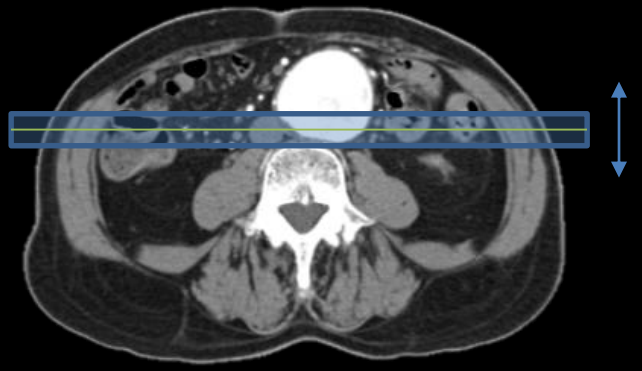


1mm MIP 10mm MIP



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

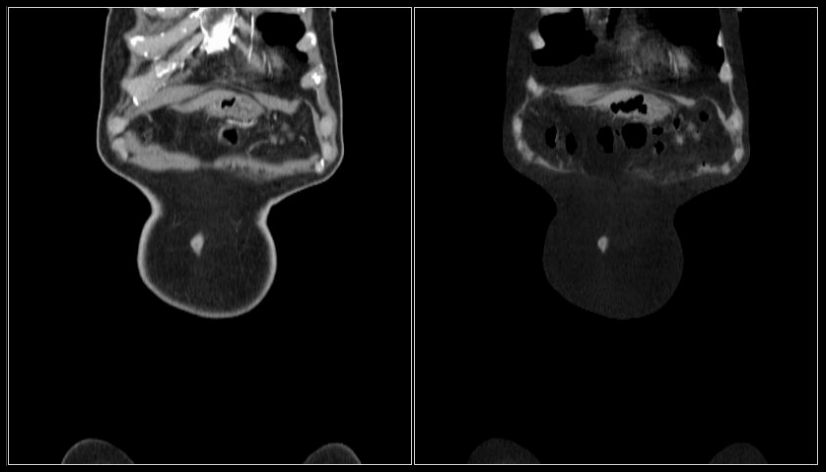
$$\max(100, 300, 400, 200) = 400$$



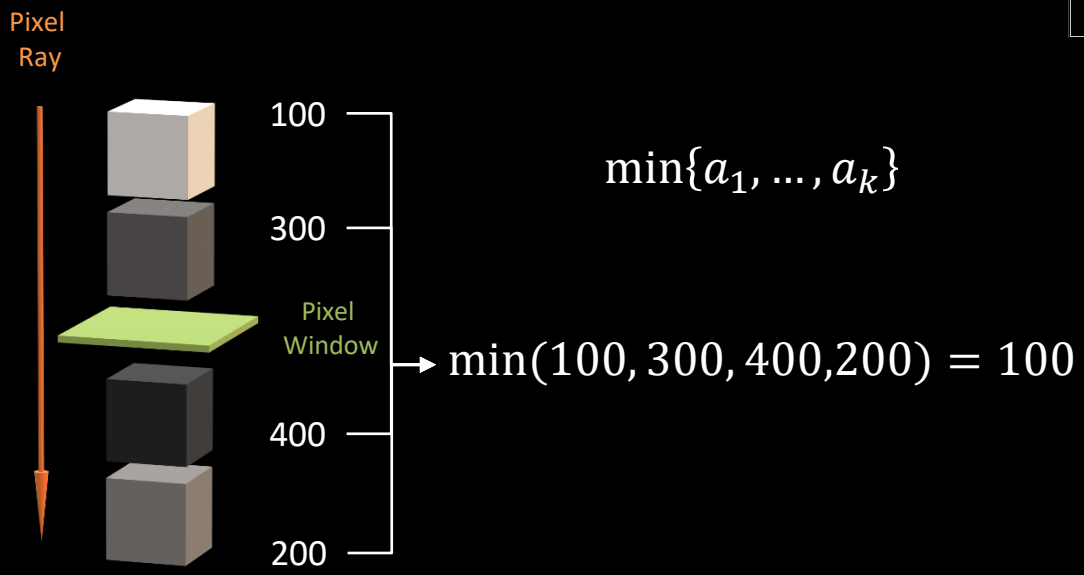
10mm

• Minimum Intensity Projection (MinIP)

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

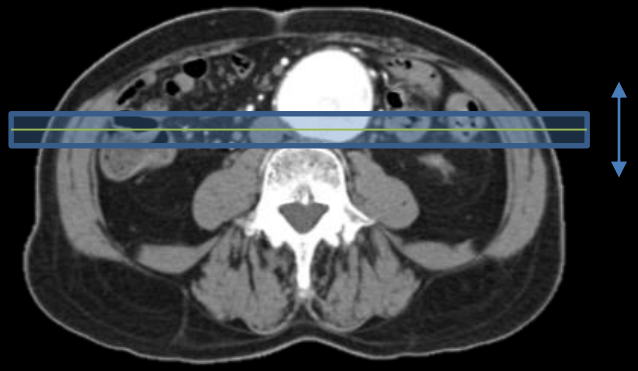


1mm MIP 10mm MIP



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

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

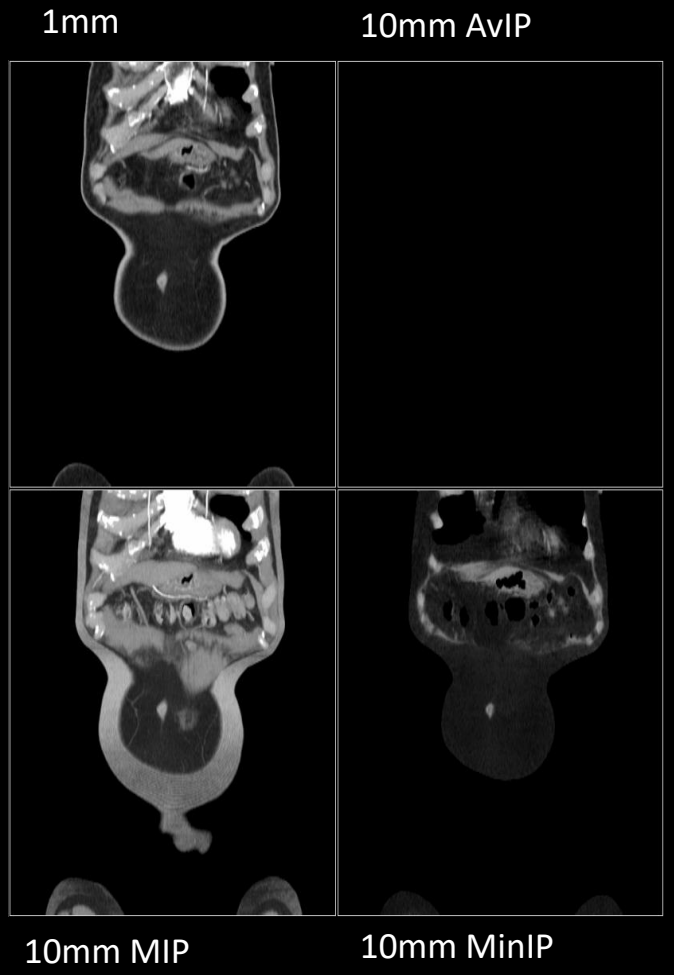


10mm

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)

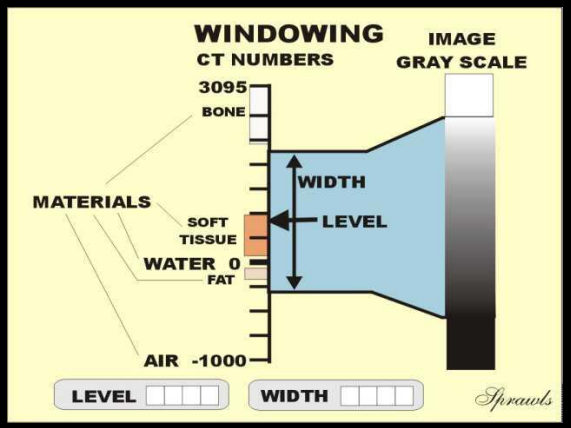
Side by Side Comparison



- # Windowing / Window Level

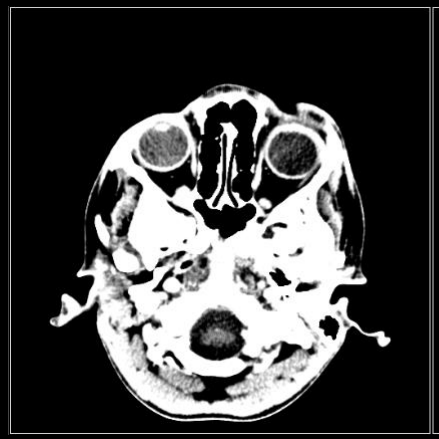
Windowing (also known as **grey-level mapping**, **contrast stretching**, **histogram modification** or **contrast enhancement)** is the process in which the greyscale components are transformed; doing this will change the representation to highlight particular structures.

Also, this allows us to convert our data our final colourspace.

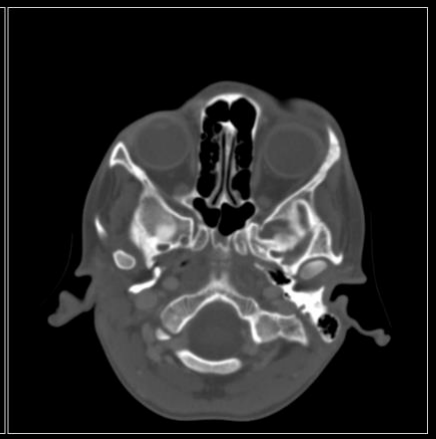


Colour space

W:80 L:40 (Brain)



W:1800 L:400 (Bone)



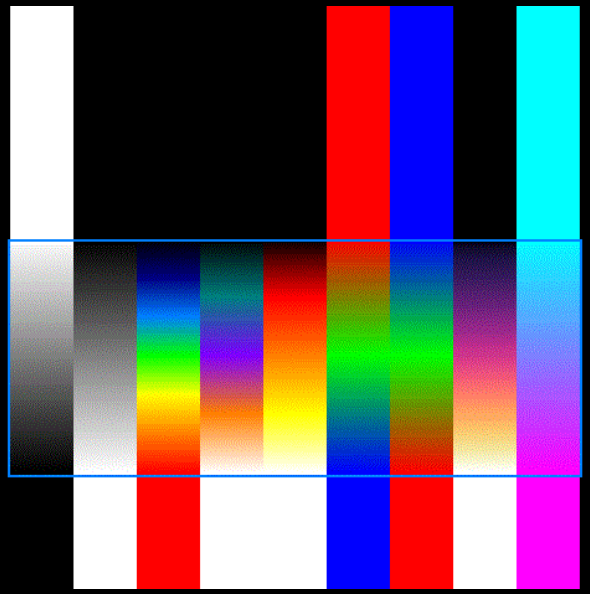
1. Understanding the requirements

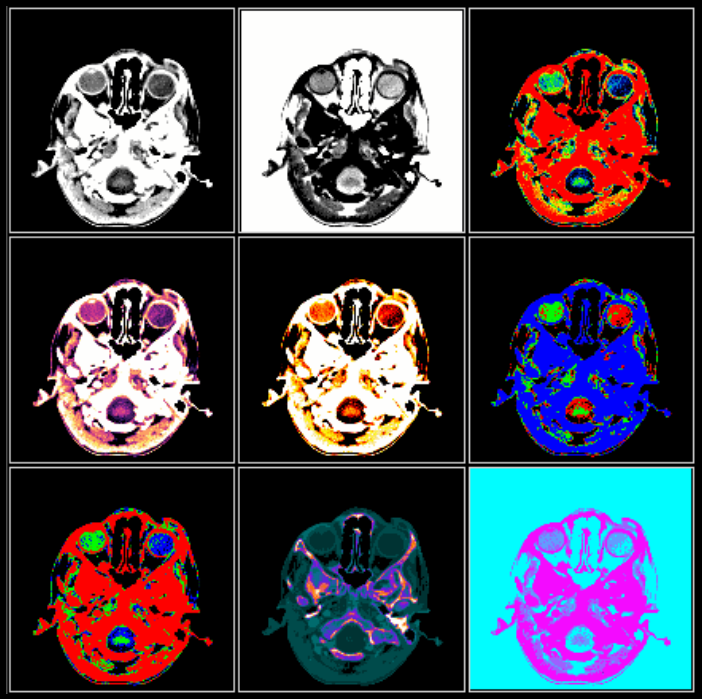
- # Colour Mapping

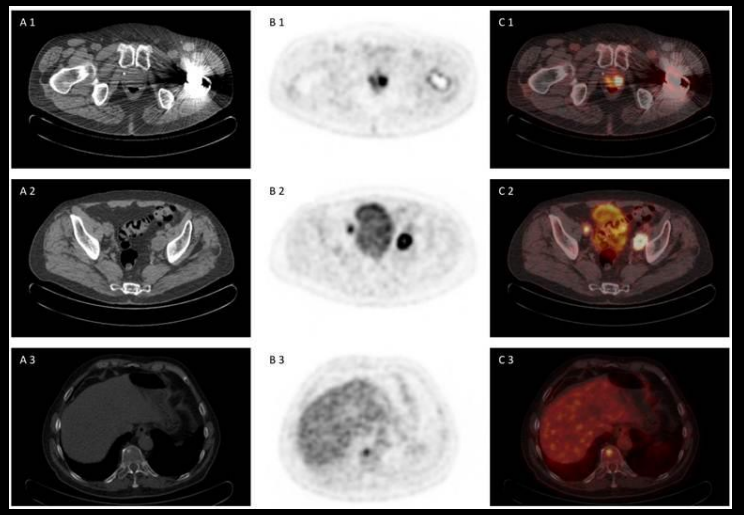
Provides additional clarity for windows of interest in the data.

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

Colour Space
[0.0-1.0]

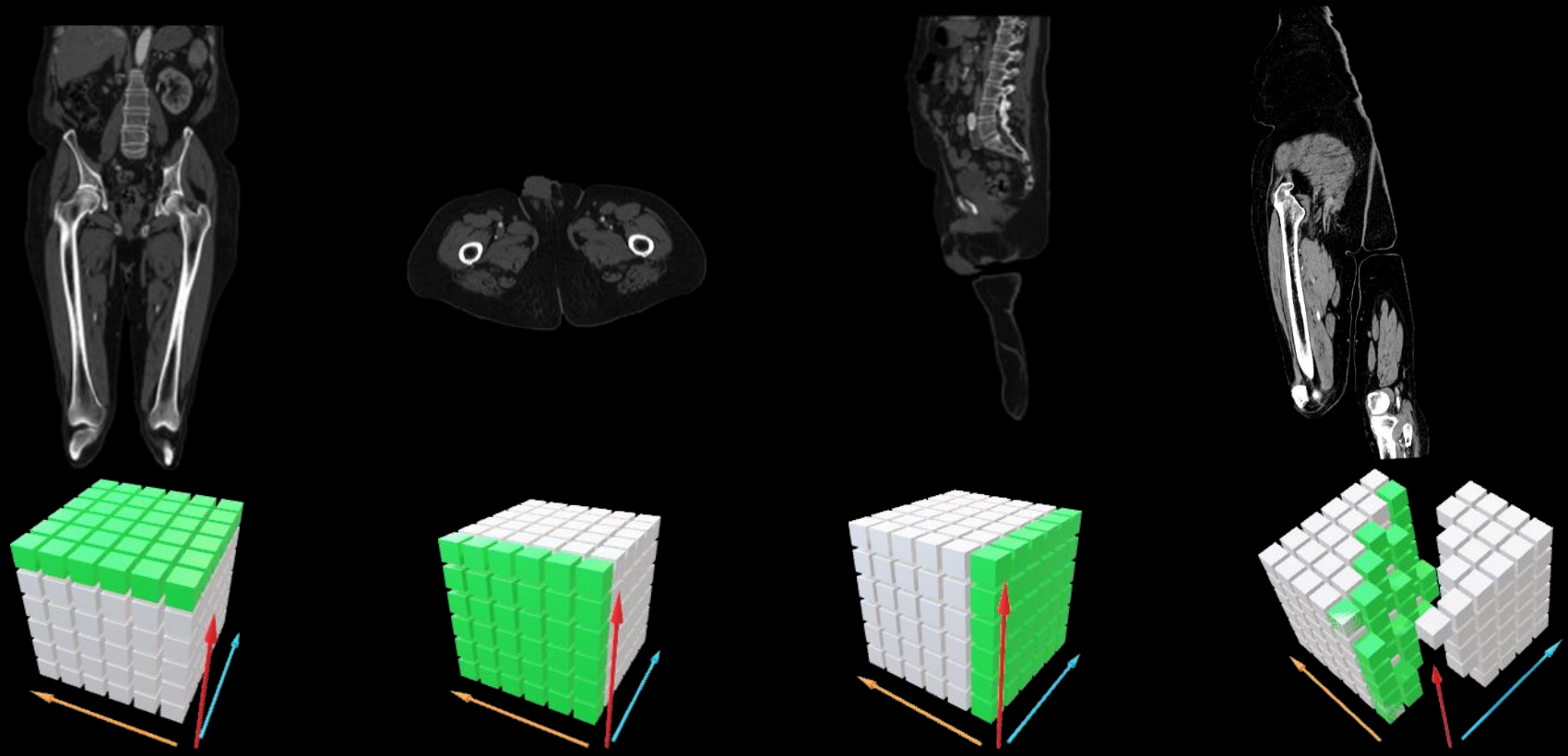






• Multiplanar Reconstruction (MPR)

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

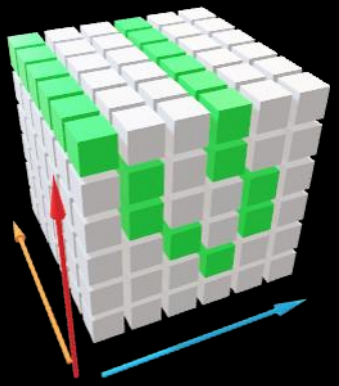
- # Curved planar Reconstruction (CPR)

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

Hyper Plane



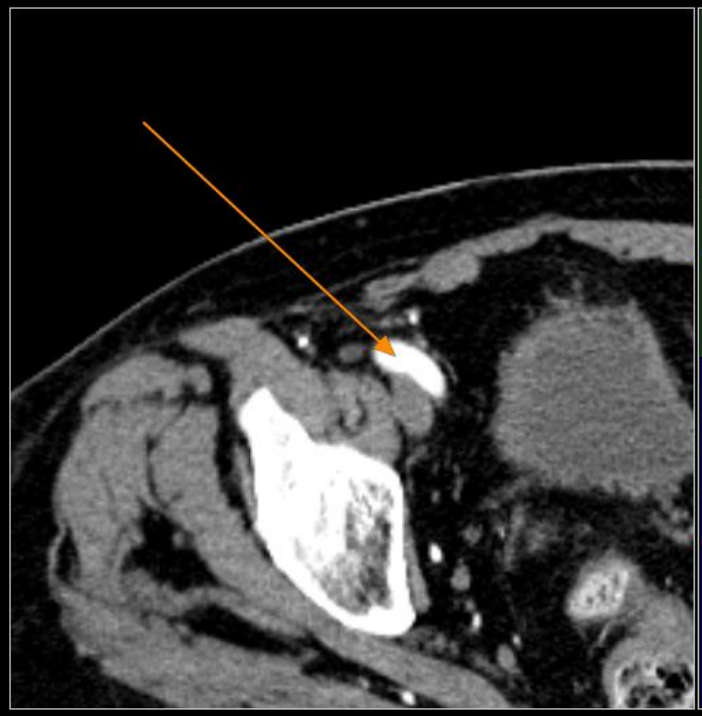
Straightened CPR

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

```
class MPRGenerator {
public:
  static void Run(
    const float windowInv, const float windowLower,
    const Vol3<ushort>& volume,
    RGBA<byte>* imageBuffer, const Vec3i& imageDIM,
    const Trans3f& imageToData) {
      std::vector<ushort> rowBlock(imageDIM[0]);
      const AABBoxf volumeBox(Vec3f::Zero(), (Vec3f)(volume.Dim() - Vec3i::One()));
      const AABBoxf volumeBoxHalf(-Vec3f::Half(), (Vec3f)(volume.Dim() - Vec3i::One()) + Vec3f::Half());

      // We must shift the image plane to the center of our slab.
      const int numOfPlanes = std::max(imageDIM[2], 1);
      const int deltaZStart = -(numOfPlanes / 2);
      const int deltaZEnd = deltaZStart + numOfPlanes;

      // Iterate down the image.(Y)
      for(int deltaY = 0; deltaY < imageDIM[1]; ++deltaY) {

        // Reset the row.
        for (auto begin = rowBlock.begin(), end = rowBlock.end(); begin != end; ++begin) {
          *begin = 0;
        }

        // Iterate through the image plane.(Z) (Starting relative to the slab)
        for(int deltaZ = deltaZStart; deltaZ < deltaZEnd; ++deltaZ) {

          // Iterate across the row.(X)
          for(int deltaX = 0; deltaX < imageDIM[0]; ++deltaX) {

            // Compute the voxel position.
            // Remember we align the image plane through the center of the slab.
            const Vec3i imagePos(deltaX, deltaY, deltaZ - imageDIM[2] / 2);
            const Vec3f voxelPos = imageToData * (Vec3f)imagePos;
            // We must be within 0.5 of an edge.
            if(AABBoxf::BoxContains(volumeBoxHalf, voxelPos)) {
              // We must clamp (to fix the 0.5 edges).
              const Vec3i voxelPosClamped = (Vec3i)volumeBox.Clamp(voxelPos);
              // Sample the value.
              const ushort dataVal = *(volume.Data() + (size_t)voxelPosClamped[0] +
              (size_t)volume.Dim()[0] * voxelPosClamped[1] +
              (size_t)volume.Dim()[0] * volume.Dim()[1] * voxelPosClamped[2]);
              rowBlock[deltaX] = std::max(rowBlock[deltaX], dataVal);
            }
          }
        }
      // Convert and copy the row block into the image buffer.
      for (auto begin = rowBlock.begin(), end = rowBlock.end(); begin != end; ++begin, ++imageBuffer) {
        const byte grayVal = std::max(0.f, std::min(255.f, ((float)(*begin) - windowLower) * windowInv));
        *imageBuffer = RGBA<byte>(grayVal, grayVal, grayVal, 255.f);
      }
    }
  }
};
```

2. Prototype MPR (Template Constant)

`MPRGenerator::Run<ProjectionMode::MIP>`

```
enum ProjectionMode { MIP, MINIP, AVIP };

class MPRGenerator {
public:
  template<ProjectionMode projectionMode>
  static void Run(
    const float windowInv, const float windowLower,
    const Vol3<ushort>& volume,
    RGBA<byte>* imageBuffer, const Vec3i& imageDIM,
    const Trans3f& imageToData) {
      std::vector<ushort> rowBlock(imageDIM[0]);
      const AABBoxf volumeBox(Vec3f::Zero(), (Vec3f)(volume.Dim() - Vec3i::One()));
      const AABBoxf volumeBoxHalf(-Vec3f::Half(), (Vec3f)(volume.Dim() - Vec3i::One()) + Vec3f::Half());

      // We must shift the image plane to the center of our slab.
      const int numOfPlanes = std::max(imageDIM[2], 1);
      const int deltaZStart = -(numOfPlanes / 2);
      const int deltaZEnd = deltaZStart + numOfPlanes;

      // Iterate down the image.(Y)
      for(int deltaY = 0; deltaY < imageDIM[1]; ++deltaY) {

        // Reset the row.
        for (auto begin = rowBlock.begin(), end = rowBlock.end(); begin != end; ++begin) {
          switch (projectionMode) {
            case ProjectionMode::MINIP:
            *begin = 0xffff;
            break;
            case ProjectionMode::MIP:
            case ProjectionMode::AVIP:
            *begin = 0;
            break;
          }
        }

        // Iterate through the image plane.(Z) (Starting relative to the slab)
        for(int deltaZ = deltaZStart; deltaZ < deltaZEnd; ++deltaZ) {

          // Iterate across the row.(X)
          for(int deltaX = 0; deltaX < imageDIM[0]; ++deltaX) {

            // Compute the voxel position.
            // Remember we align the image plane through the center of the slab.
            const Vec3i imagePos(deltaX, deltaY, deltaZ - imageDIM[2] / 2);
            const Vec3f voxelPos = imageToData * (Vec3f)imagePos;
            // We must be within 0.5 of an edge.
            if(AABBoxf::BoxContains(volumeBoxHalf, voxelPos)) {
              // We must clamp (to fix the 0.5 edges).
              const Vec3i voxelPosClamped = (Vec3i)volumeBox.Clamp(voxelPos);
            // Sample the value.
            const ushort dataVal = *(volume.Data() + (size_t)voxelPosClamped[0] +
              (size_t)volume.Dim()[0] * voxelPosClamped[1] +
              (size_t)volume.Dim()[0] * volume.Dim()[1] * voxelPosClamped[2]);


            switch (projectionMode) {
              case ProjectionMode::MIP:
              rowBlock[deltaX] = std::max(rowBlock[deltaX], dataVal);
              break;
              case ProjectionMode::MINIP:
              rowBlock[deltaX] = std::min(rowBlock[deltaX], dataVal);
              break;
              case ProjectionMode::AVIP:
              rowBlock[deltaX] += rowBlock[deltaX];
              break;
            }

          }
        }
      }

      if(projectionMode == ProjectionMode::AVIP) {
        for (auto begin = rowBlock.begin(), end = rowBlock.end(); begin != end; ++begin) {
          *begin / imageDIM[2];
        }
      }

      // Convert and copy the row block into the image buffer.
      for (auto begin = rowBlock.begin(), end = rowBlock.end(); begin != end; ++begin, ++imageBuffer) {
        const byte grayVal = std::max(0.f, std::min(255.f, ((float)(*begin) - windowLower) * windowInv));
        *imageBuffer = RGBA<byte>(grayVal, grayVal, grayVal, 255.f);
      }
    }
  }
};
```

2. Prototype MPR (Polymorphic Templates)

```cpp
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 {}
};
```

```cpp
#include "Accumulator.cpp"
#include <limits>


template<class T>
class MinIPAccumulator : public Accumulator<T> {
public:
  inline T Sample(const T v, const T r) const {
    return std::min(v, r);
  }

  template<class Iter>
  inline void PreSample(Iter begin, const Iter end) const {
    for (; begin != end; ++begin) {
      *begin = std::numeric_limits<T>::max();
    }
  }
};
```

```cpp
#include "Accumulator.cpp"
#include <limits>

template<class T>
class MIPAccumulator : public Accumulator<T> {
public:
  inline T Sample(const T v, const T r) const {
    return std::max(v, r);
  }

  template<class Iter>
  inline void PreSample(Iter begin, const Iter end) const {
    for (; begin != end; ++begin) {
      *begin = std::numeric_limits<T>::min();
    }
  }
};
```

```cpp
#include "Accumulator.cpp"

template <class T>
class AvIPAccumulator : public Accumulator<T> {
public:
  inline T Sample(const T v, const T r) const {
    return v + r;
  }

  template<class Iter>
  inline void PostSample(Iter begin, const Iter end) const {
    const int len = end - begin;
    for (; begin != end; ++begin) {
      *begin /= len;
    }
  }
};
```

2. Prototype MPR (Polymorphic Templates)

```
MPRGenerator::Run<MIPAccumulator>
```

```
class MPRGenerator {
public:
  template<class Accumulator>
  static void Run(
    const float windowInv, const float windowLower,
    const Vol3<ushort>& volume,
    RGBA<byte>* imageBuffer, const Vec3i& imageDIM,
    const Trans3f& imageToData) {
      std::vector<ushort> rowBlock(imageDIM[0]);
      const AABBoxf volumeBox(Vec3f::Zero(), (Vec3f)(volume.Dim() - Vec3i::One()));
      const AABBoxf volumeBoxHalf(-Vec3f::Half(), (Vec3f)(volume.Dim() - Vec3i::One()) + Vec3f::Half()
      const Accumulator accumulator;

      // We must shift the image plane to the center of our slab.
      const int numOfPlanes = std::max(imageDIM[2], 1);
      const int deltaZStart = -(numOfPlanes / 2);
      const int deltaZEnd = deltaZStart + numOfPlanes;

      // Iterate down the image.(Y)
      for(int deltaY = 0; deltaY < imageDIM[1]; ++deltaY) {

        // Reset the row accumulators.
        accumulator.PreSample(rowBlock.begin(), rowBlock.end());

        // Iterate through the image plane.(Z) (Starting relative to the slab)
        for(int deltaZ = deltaZStart; deltaZ < deltaZEnd; ++deltaZ) {
          for(int deltaX = 0; deltaX < imageDIM[0]; ++deltaX) {

            // Compute the voxel position.
            // Remember we align the image plane through the center of the slab.
            const Vec3i imagePos(deltaX, deltaY, deltaZ - imageDIM[2] / 2);
            const Vec3f voxelPos = imageToData * (Vec3f)imagePos;
            // We must be within 0.5 of an edge.
            if(AABBoxf::BoxContains(volumeBoxHalf, voxelPos)) {
              // We must clamp (to fix the 0.5 edges).
              const Vec3i voxelPosClamped = (Vec3i)volumeBox.Clamp(voxelPos);
              // Sample the value.
              const ushort dataVal = *(volume.Data() + (size_t)voxelPosClamped[0] +
              (size_t)volume.Dim()[0] * voxelPosClamped[1] +
              (size_t)volume.Dim()[0] * volume.Dim()[1] * voxelPosClamped[2]);

              rowBlock[deltaX] = accumulator.Sample(rowBlock[deltaX], dataVal);
            }
          }
        }

        // Complete the row accumulators.
        accumulator.PostSample(rowBlock.begin(), rowBlock.end());

        // Convert and copy the row block into the image buffer.
        for (auto begin = rowBlock.begin(), end = rowBlock.end(); begin != end; ++begin, ++imageBuffer) {
          const byte grayVal = std::max(0.f, std::min(255.f, ((float)(*begin) - windowLower) * windowInv));
          *imageBuffer = RGBA<byte>(grayVal, grayVal, grayVal, 255.f);
        }
      }
    }
};
```

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
```

```
#include "Converter.cpp"


template<class T>
class WLConverter : public Converter<T, float> {
  float m_windowInv, m_lower_window_limit;

public:
  WLConverter(const float window, const float level):
   m_windowInv(1.0f / window),
   m_lower_window_limit(level - window / 2) {}
  float Convert(const T v) const {
   return ((float)v - m_lower_window_limit) * m_windowInv;
  }
};
```

```
#include "Converter.cpp"
#include "../RGBA.cpp"

template<class T>
class ColourMapConverter : public Converter<float, RGBA<T>> {
private:
  const std::map<float, RGBA<T>> m_colourLUT;
  const float m_scalar;
public:
  ColourMapConverter(const std::map<float, RGBA<T>>& colourmap):
  m_colourLUT(colourmap.begin(), colourmap.end()),
   m_scalar(!colourmap.empty()
   ? 1.f / ((*--colourmap.end()).first - (*colourmap.begin()).first)
   : 1.f) {}
  RGBA<T> Convert(const float v) const {
     const float vClamped = std::min(1.f, std::max(0.f, v));
     const auto c0Iter = m_colourLUT.lower_bound(vClamped);
     const auto c1Iter = m_colourLUT.upper_bound(vClamped);
     const float t = (vClamped - (*c0Iter).first) * m_scalar;

     return mix((*c0Iter).second, (*c1Iter).second, t);
  }
};
```

```
#include "Converter.cpp"

template<class Tin, class Tout, class Cin, class Cout>
class InlineConverter : public Converter<Tin, Tout> {
private:
  Cin m_in;
  Cout m_out;
public:
    InlineConverter(const Cin in, const Cout out):
    m_in(in), m_out(out) {}
    Tout Convert(const Tin v) const {
      return m_out.Convert(m_in.Convert(v));
    }
};
```

2. Prototype MPR (Polymorphic Templates)

```
template<class DataType>
class Vol3Sampler {
public:
  virtual size_t GetOffsetAtPosition(const Vol3<DataType>& vol, const Vec3i& pos) const = 0;

  virtual int GetXPositionAtOffset(const Vol3<DataType>& vol, const size_t offset) const = 0;

  virtual int GetYPositionAtOffset(const Vol3<DataType>& vol, const size_t offset)  const = 0;

  virtual int GetZPositionAtOffset(const Vol3<DataType>& vol, const size_t offset)  const = 0;

  inline const DataType* GetPtrAtPosition(const Vol3<DataType>& vol, Vec3i pos) const {
    assert(vol.Data() != nullptr);
    return vol.Data() + GetOffsetAtPosition(vol, pos);
  }

  inline DataType* GetPtrAtPosition(Vol3<DataType>& vol, Vec3i pos) {
    assert(vol.Data() != nullptr);
    return vol.Data() + GetOffsetAtPosition(vol, pos);
  }

  inline const DataType& GetDataAtPosition(const Vol3<DataType>& vol, const Vec3i& pos) const {
    return *GetPtrAtPosition(vol, pos);
  }

  inline DataType& GetDataAtPosition(Vol3<DataType>& vol, const Vec3i& pos) {
    return *GetPtrAtPosition(vol, pos);
  }

  inline const DataType& GetDataAtOffset(const Vol3<DataType>& vol, const size_t offset) const {
    assert(vol.Data() != nullptr);
    return *(vol.Data() + offset);
  }

  inline DataType& GetDataAtOffset(Vol3<DataType>& vol, const size_t offset) {
    assert((size_t)vol.Dim()[0] * vol.Dim()[1] * vol.Dim()[2] > offset);
    return *(vol.Data() + offset);
  }

  inline size_t GetOffsetAtPtr(const Vol3<DataType>& vol, const DataType* const ptr) const {
    assert((size_t)vol.Dim()[0] * vol.Dim()[1] * vol.Dim()[2] > (size_t)(ptr - vol.Data()));
    return (size_t)(ptr - vol.Data());
  }
```

```
  inline Vec3i GetPositionAtOffset(const Vol3<DataType>& vol, const size_t offset) const {
    return Vec3i(
      GetXPositionAtOffset(vol, offset),
      GetYPositionAtOffset(vol, offset),
      GetZPositionAtOffset(vol, offset)
    );
  }

  inline Vec3i GetPositionAtPtr(const Vol3<DataType>& vol, const DataType* const ptr) const {
    return GetPositionAtOffset(GetOffsetAtPtr(vol, ptr));
  }
};
```

```
#include "../GlobalDefs.h"
#include "Vol3Sampler.cpp"

template<class DataType>
class Vol3SamplerLinear : public Vol3Sampler<DataType> {
public:
  inline size_t GetOffsetAtPosition(const Vol3<DataType>& vol, const Vec3i& pos) const {
    assert(pos[0] < pos[0] >= 0);
    assert(pos[0] < vol.Dim()[0]);
    assert(pos[1] < pos[1] >= 0);
    assert(pos[1] < vol.Dim()[1]);
    assert(pos[2] < pos[2] >= 0);
    assert(pos[2] < vol.Dim()[2]);

    return (size_t)pos[0] +
    (size_t)vol.Dim()[0] * pos[1] +
    (size_t)vol.Dim()[0] * vol.Dim()[1] * pos[2];
  }

  inline int GetXPositionAtOffset(const Vol3<DataType>& vol, const size_t offset) const {
    return (int)(offset % vol.Dim()[0]);
  }

  inline int GetYPositionAtOffset(const Vol3<DataType>& vol, const size_t offset) const {
     return (int)(offset / vol.Dim()[0]) % vol.Dim()[1];
  }

  inline int GetZPositionAtOffset(const Vol3<DataType>& vol, const size_t offset) const {
    return (int)(offset / vol.Dim()[0] / vol.Dim()[1]);
  }
};
```

2. Prototype MPR (Polymorphic Templates)

```
4    template<class DataType, class VolType, class VolSampler>
5    class InterpKernel {
6    protected:
7        const VolSampler m_sampler;
8    public:
9      inline virtual DataType Sample(const VolType& volume, const Vec3f& position) const = 0;
10   };
11
```

```
4    #include "InterpKernel.cpp"
5
6    template<class DataType, class VolType, class VolSampler>
7    class LIInterpKernel : public InterpKernel<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, 0));
18       const float v5 = (float)this->m_sampler.GetDataAtPosition(volume, (Vec3i)position + Vec3i(1, 0, 0));
19       const float v6 = (float)this->m_sampler.GetDataAtPosition(volume, (Vec3i)position + Vec3i(0, 1, 0));
20       const float v7 = (float)this->m_sampler.GetDataAtPosition(volume, (Vec3i)position + Vec3i(1, 1, 0));
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   };
```

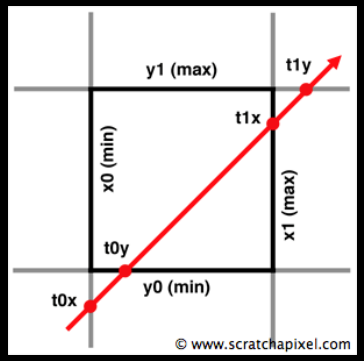
```
4    #include "InterpKernel.cpp"
5
6    template<class DataType, class VolType, class VolSampler>
7    class NNInterpKernel : public InterpKernel<DataType, VolType, VolSampler> {
8    public:
9      inline DataType Sample(const VolType& volume, const Vec3f& position) const {
10       return this->m_sampler.GetDataAtPosition(volume,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

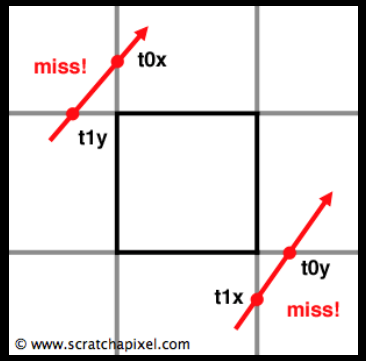
2. Prototype MPR (Polymorphic Templates)

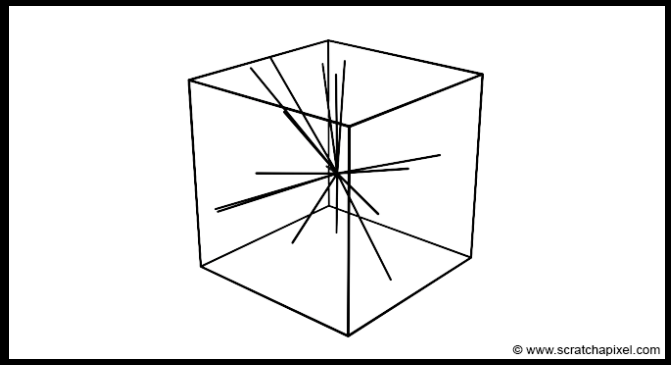
```
class MPRGenerator {
public:
  template<class VolType, class DataType, class PixelType, class Accumulator, class Converter, class InterpKernal>
  static void Run(
    const VolType& volume,
    PixelType* imageBuffer,
    const Vec3i& imageDIM,
    const Trans3f& imageToData,
    const Converter& converter) {
      const Accumulator accumulator;
      const InterpKernal interpKernal;
      std::vector<DataType> rowBlock(imageDIM[0]);
      const AABBoxf volumeBox(Vec3f::Zero(), (Vec3f)(volume.Dim() - Vec3i::One()));
      const AABBoxf volumeBoxHalf(-Vec3f::Half(), (Vec3f)(volume.Dim() - Vec3i::One()) + Vec3f::Half());

      // We must shift the image plane to the center of our slab.
      const int numOfPlanes = std::max(imageDIM[2], 1);
      const int deltaZStart = -(numOfPlanes / 2);
      const int deltaZEnd = deltaZStart + numOfPlanes;

      // Iterate down the image.(Y)
      for(int deltaY = 0; deltaY < imageDIM[1]; ++deltaY) {

        // Reset the row accumulators.
        accumulator.PreSample(rowBlock.begin(), rowBlock.end());

        // Iterate through the image plane.(Z) (Starting relative to the slab)
        for(int deltaZ = deltaZStart; deltaZ < deltaZEnd; ++deltaZ) {

          // Iterate across the row.(X)
          for(int deltaX = 0; deltaX < imageDIM[0]; ++deltaX) {
            // Compute the voxel position.
            // Remember we align the image plane through the center of the slab.
            const Vec3i imagePos(deltaX, deltaY, deltaZ);
            const Vec3f voxelPos = imageToData * (Vec3f)imagePos;

            // We must be within 0.5 of an edge.
            if(AABBoxf::BoxContains(volumeBoxHalf, voxelPos)) {
              // We must be within 0.5 of an edge.
              const Vec3f voxelPosClamped = volumeBox.Clamp(voxelPos);
              // Sample the value.
              const DataType dataVal = interpKernal.Sample(volume, voxelPosClamped);
              rowBlock[deltaX] = accumulator.Sample(rowBlock[deltaX], dataVal);
            }
          }
        }

        // Complete the row accumulators.
        accumulator.PostSample(rowBlock.begin(), rowBlock.end());

        // Convert and copy the row block into the image buffer.
        for (auto begin = rowBlock.begin(), end = rowBlock.end(); begin != end; ++begin, ++imageBuffer) {
          *imageBuffer = converter.Convert(*begin);
        }
      }
    }
```

3. Generic Volume Renderer

- Further reduce operations by removing certain checks per pixel.








```
(m_volumeBox.IntersectRayAlternative(depthPos, vXDirInv, vXDirInvSign, xMinInside, xMaxInside)
```

- Generalization to support multiple visualization techniques, such as CPR.

```
// Provides a mechanisum that iterates through the image row by row. The reference vectors
// defines how each row is mapped into the volume data space for sampling.
class RowTransformer {
public:
  // Set the row, compute any necessary values for RowPos, RowDir and DepthDir in this call
  virtual void GetRowDetails(const int j, Vec3f& pos, Vec3f& rowDir, Vec3f& depthDir) const = 0;
  // Get the image space dimensions.
  virtual const Vec3i& GetImageDIM() const = 0;
};
```

3. Generic Volume Renderer

- General Multithreading support.

```
4     class ThreadedRendererThread {
5     public:
6       virtual void operator()(int threadIdx, int numOfThreads) const = 0;
7     };
8
```

```
6     class ThreadedRenderer {
7     public:
8       ThreadedRenderer(const ThreadedRendererThread& renderer, const int numOfThreads);
9       void Run();
10    };
```

3. Generic Volume Renderer

```
template<class DataType, class ImageType, class VolType, class RowTransformer, class Accumulator, class Converter, class InterpKernal>
class SlabGenerator : public ThreadedRendererThread {
private:
  const VolType& m_volume;
  ImageType*const m_imageBuffer;
  const Converter m_converter;
  const RowTransformer m_rowTransformer;
  const AABBoxf m_volumeBox;
  const AABBoxf m_volumeBoxHalf;
  const Accumulator m_accumulator;
  const InterpKernal m_interpKernal;
  const Vec3i m_imageDIM;
  const int m_numOfPlanes;
  const int m_deltaZStart;
  const int m_deltaZEnd;
public:
  SlabGenerator(
    const VolType& volume, const Converter& converter,
    ImageType*const imageBuffer, const RowTransformer& rowTransformer):
   m_volume(volume),   m_converter(converter),   m_imageBuffer(imageBuffer),
   m_rowTransformer(rowTransformer),
   m_volumeBox(Vec3f::Zero(), (Vec3f)(volume.Dim() - Vec3i::One())),
   m_volumeBoxHalf(-Vec3f::Half(), (Vec3f)(volume.Dim() - Vec3i::One()) + Vec3f::Half()),
   m_imageDIM(rowTransformer.GetImageDIM()),
   m_numOfPlanes(std::max(m_imageDIM[2], 1)),   m_deltaZStart(-(m_numOfPlanes / 2)),   m_deltaZEnd(m_deltaZStart + m_numOfPlanes)
  {}
  void operator() (const int iThread, const int numThreads) const {
      ImageType* imageBuffer = m_imageBuffer + iThread;
      std::vector<DataType> rowBlock(m_imageDIM[0]);

      // Iterate down the image.(Y)
      for(int deltaY = iThread; deltaY < m_imageDIM[1]; deltaY += numThreads) {
        // Get Bi-Normal/Normal/Position for the first pixel in the row to volume data space.
        Vec3f vPos, vXDir, vZDir;
        m_rowTransformer.GetRowDetails(deltaY, vPos, vXDir, vZDir);

        // Pre-compute the inverse ray for intersection tests.
        Vec3f vXDirInv; Vec3i vXDirInvSign;
        AABBoxf::SignedInverseRay(vXDir, vXDirInv, vXDirInvSign);

        // Reset the row accumulators.
        m_accumulator.PreSample(rowBlock.begin(), rowBlock.end());

        // Iterate through the image plane.(Z) (Starting relative to the slab)
        for(int deltaZ = m_deltaZStart; deltaZ < m_deltaZEnd; ++deltaZ) {

          const Vec3f depthPos = vPos + vZDir * (float)deltaZ;

          // Iterate across the row.(X)
          for(int deltaX = 0; deltaX < m_imageDIM[0]; ++deltaX) {

            float xMinEdge, xMaxEdge, xMinInside, xMaxInside;
            // Intersect and test we are < 0.5 of the volume edges. If not nothing to do.
            if (m_volumeBoxHalf.IntersectRayAlternative(depthPos, vXDirInv, vXDirInvSign, xMinEdge, xMaxEdge)) {
              // Clamp the outer volume Min/Max to the image range.
              xMinEdge = std::max(xMinEdge, 0.f);
              xMaxEdge = std::min(xMaxEdge, (float)m_imageDIM[0]);

              // Get the starting ImageX.
              int deltaX = (int)fabsf(xMinEdge);

              // Intersect the inner volume bounds where interpolation is allowed.
              if (m_volumeBox.IntersectRayAlternative(depthPos, vXDirInv, vXDirInvSign, xMinInside, xMaxInside)) {
                // Clamp the inner volume Min/Max to the max image range.
                xMinInside = std::min(xMinInside, (float)m_imageDIM[0]);
                xMaxInside = std::min(xMaxInside, (float)m_imageDIM[0]);

                // Cycle though the width minimising if checks, we do this by computing the X intersection.
                // We loop though the left edge, then the middle, and finally right edge.
                // Within 0.5 on the left of the volume box - Interpolation is *NOT* safe. Position must be clamped.
                for (; deltaX < xMinInside; ++deltaX) {
                  const Vec3f finalPos = depthPos + vXDir * (float)deltaX;
                  const Vec3f clampedPos = m_volumeBox.Clamp(finalPos);
                  // Sample the value.
                  const DataType dataVal = m_interpKernal.Sample(m_volume, clampedPos);
                  rowBlock[deltaX] = m_accumulator.Sample(rowBlock[deltaX], dataVal);
                }

                // Inside the volume box - Interpolation is safe...
                for (; deltaX < xMaxInside; ++deltaX) {
                  const Vec3f finalPos = depthPos + vXDir * (float)deltaX;
                  // Sample the value.
                  const DataType dataVal = m_interpKernal.Sample(m_volume, finalPos);
                  rowBlock[deltaX] = m_accumulator.Sample(rowBlock[deltaX], dataVal);
                }
              }
            }
          }
          // Within 0.5 on the right of the volume box - Interpolation is *NOT* safe. Position must be clamped.
          for (; deltaX < xMaxEdge; ++deltaX) {
            const Vec3f finalPos = depthPos + vXDir * (float)deltaX;
            const Vec3f clampedPos = m_volumeBox.Clamp(finalPos);
            // Sample the value.
            const DataType dataVal = m_interpKernal.Sample(m_volume, clampedPos);
            rowBlock[deltaX] = m_accumulator.Sample(rowBlock[deltaX], dataVal);
          }
        }
      }
      // Complete the row accumulators.
      m_accumulator.PostSample(rowBlock.begin(), rowBlock.end());

      // Convert and copy the row block into the image buffer.
      for (auto begin = rowBlock.begin(), end = rowBlock.end(); begin != end; ++begin, ++imageBuffer) {
        *imageBuffer = m_converter.Convert(*begin);
      }
    }
  }
};
```

THANK YOU!
Questions?