3D Volume Viewer Overview

2019.07.08

Rapiscan Systems

Karthik Bala
KBVIS Technologies Pvt. Ltd.
3d@kbvis.com

DAY 1

Setup and Resources

- WebGL 3D DICOM Viewer (and test DICOM volume)

http://kbvis.com/downloads/rapiscan-papaya-viewer.zip

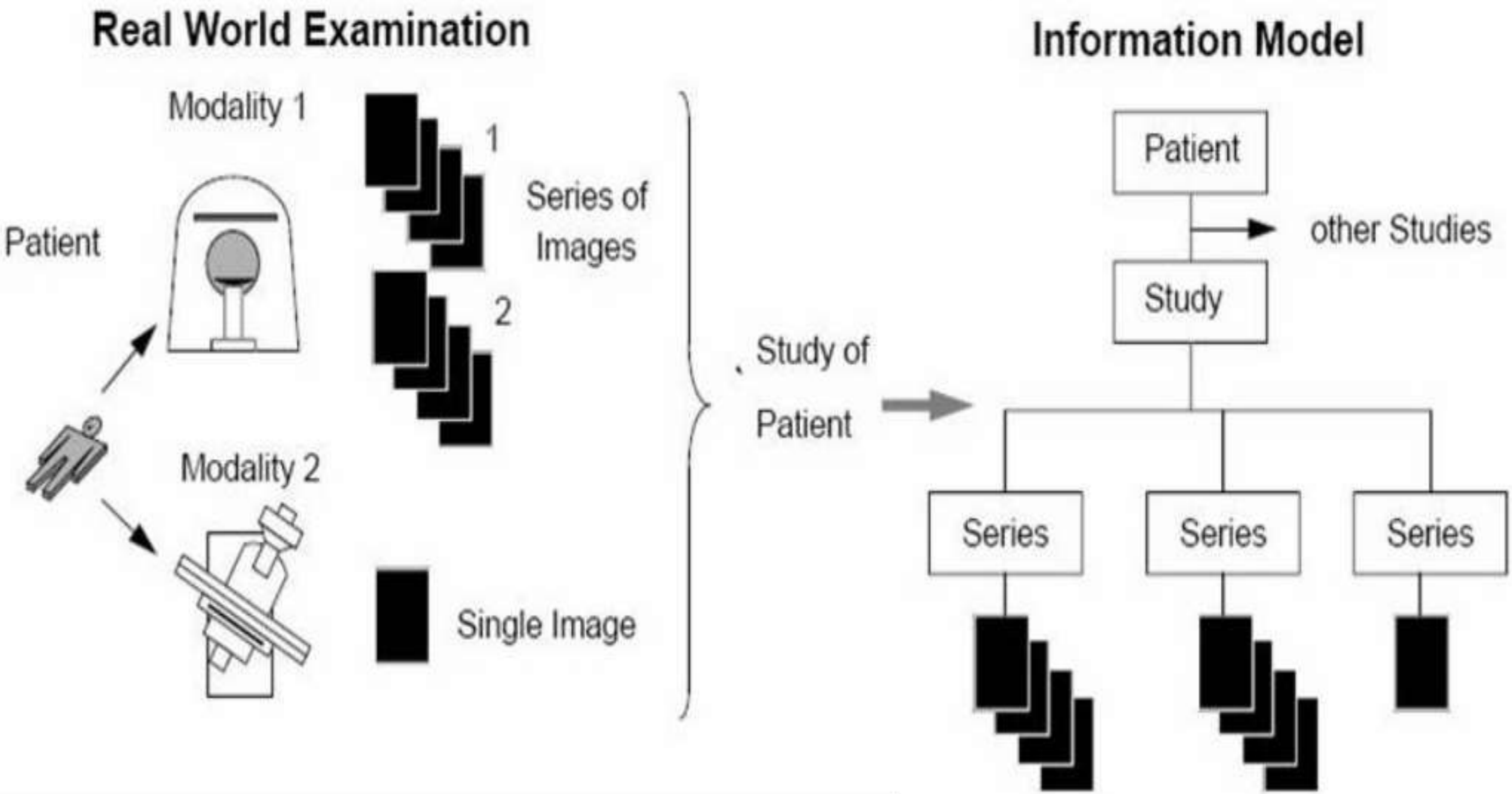
- **Digital Imaging and Communications in Medicine**
- Standard for the communication and management of medical imaging across devices, workstations, networks, PACS
- Vendor-independent, where DICOM conformant
- Derivations:
  - DICONDE – Non destructive testing
  - DICOS – Security applications

Equipment Types

- CT (computed tomography)
- MRI (magnetic resonance imaging)
- Ultrasound
- X-ray
- Fluoroscopy
- Angiography
- Mammography
- Breast tomosynthesis
- PET (positron emission tomography)
- SPECT (single photon emission computed tomography)
- Endoscopy
- Microscopy
- Whole slide imaging
- OCT (optical coherence tomography)
- PACS (picture archiving and communication systems)
- Image viewers and display stations
- CAD (computer-aided detection/diagnosis systems)
- 3D visualization systems
- Clinical analysis applications
- Image printers
- Film scanners
- Media burners (that export DICOM files onto CDs, DVDs, etc)
- Media importers (that import DICOM files from CDs, DVDs, USBs, etc)
- RIS (radiology information systems)
- VNA (vendor-neutral archives)
- EMR (electronic medical record) systems

Imaging Studies

- **Study** – Patient scan data consisting of one or more image series
- **Series** – Set of images from a single scan, or secondary capture.
- **Image –** represents single section or slice through the anatomy
- **Volume Image** – Series acquired from a volume scan. Constituent images have consistent spacing, orientation


Real World Examination
Modality 1
1
Series of Images
2
Patient
Modality 2
Single Image
Study of Patient
Information Model
Patient
other Studies
Study
Series
Series
Series

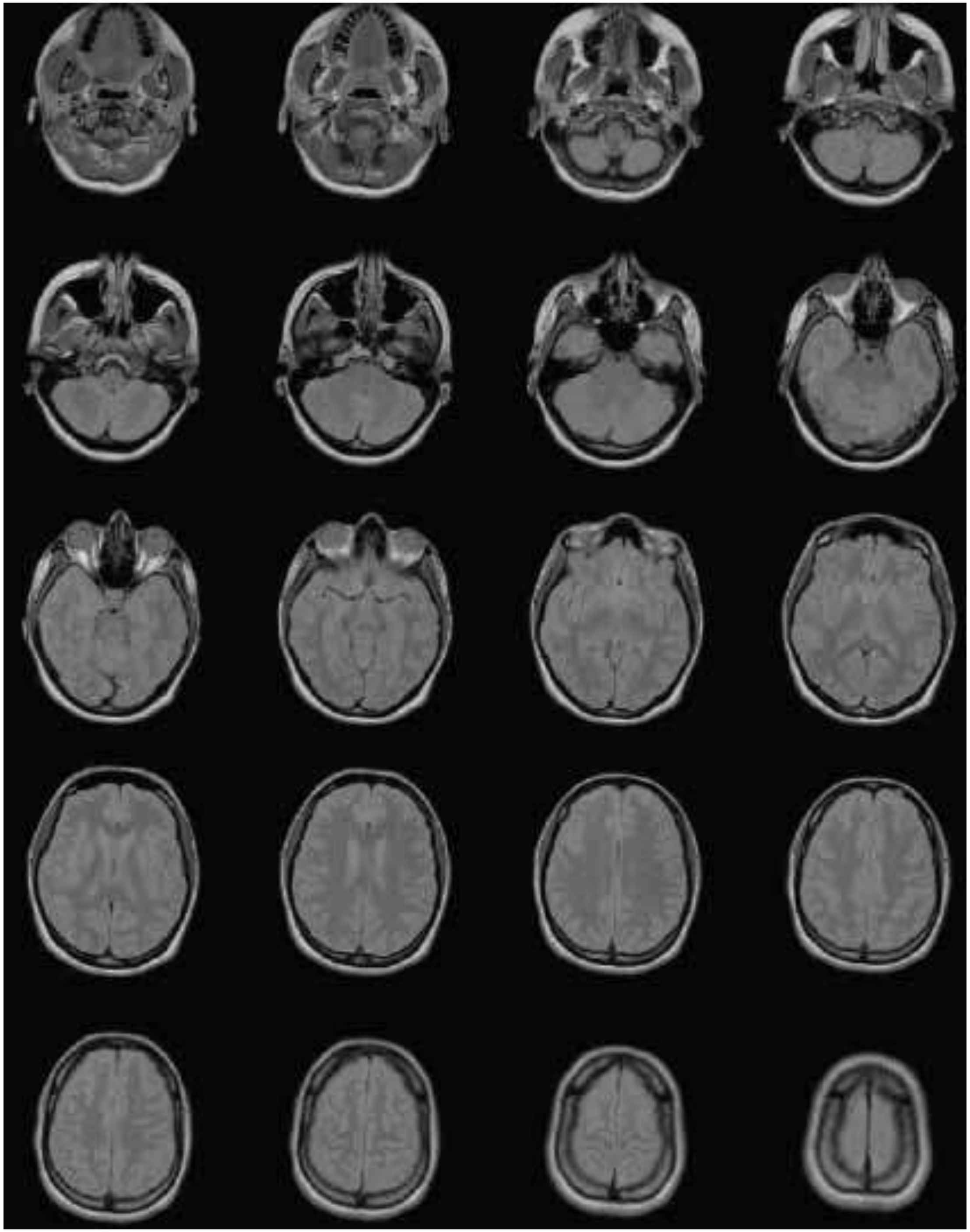
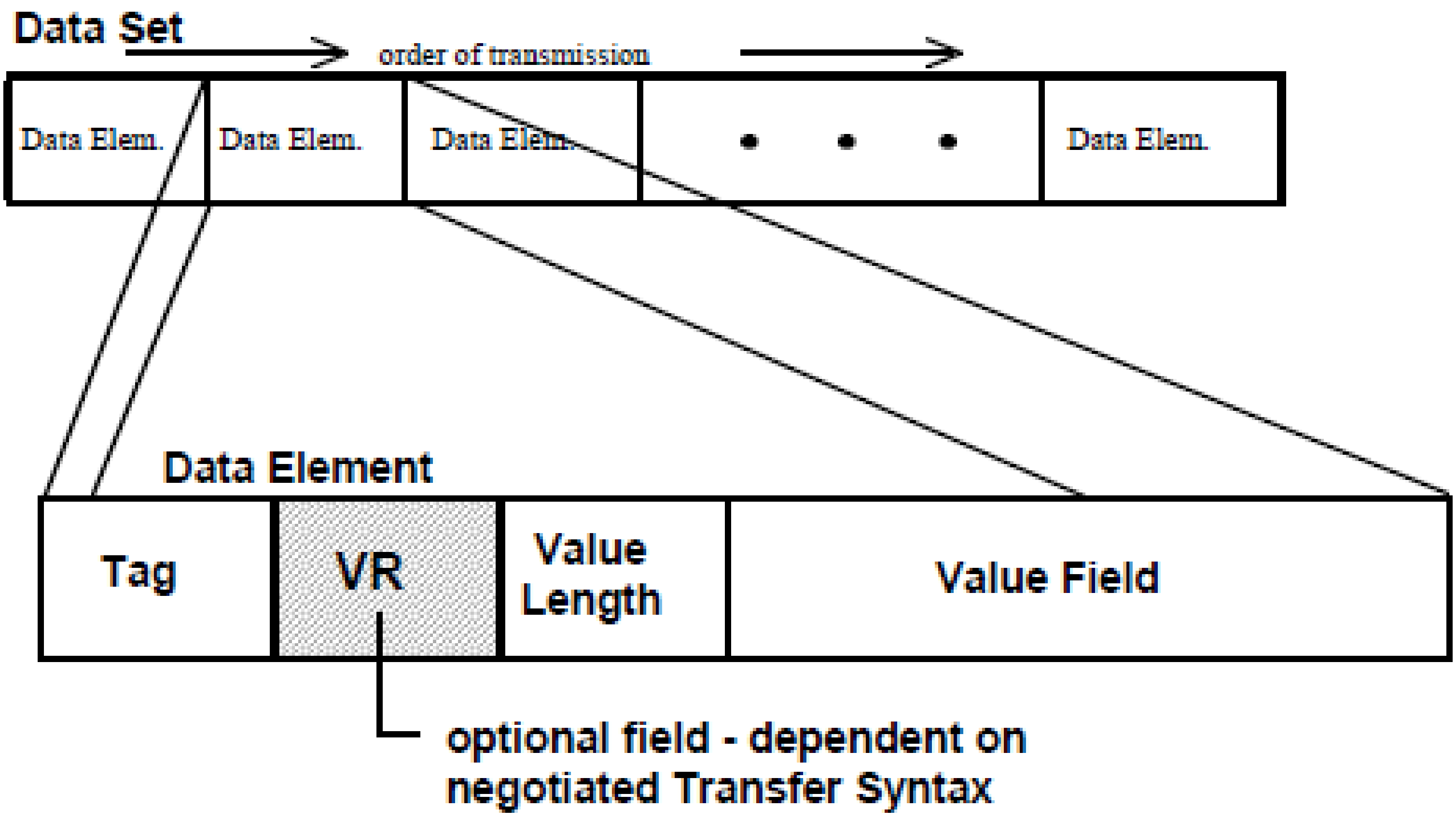


Image Series

DICOM Parsing



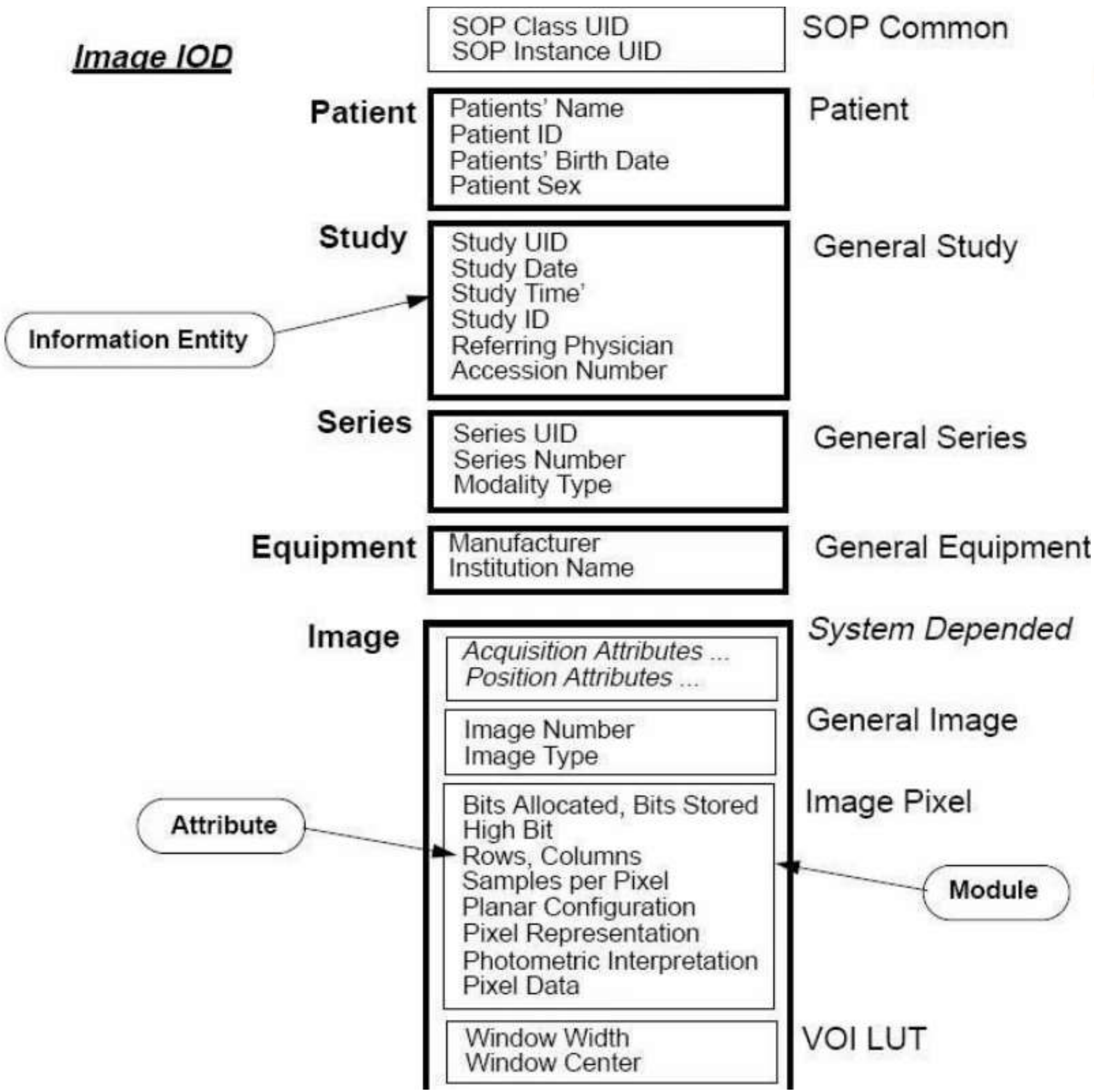
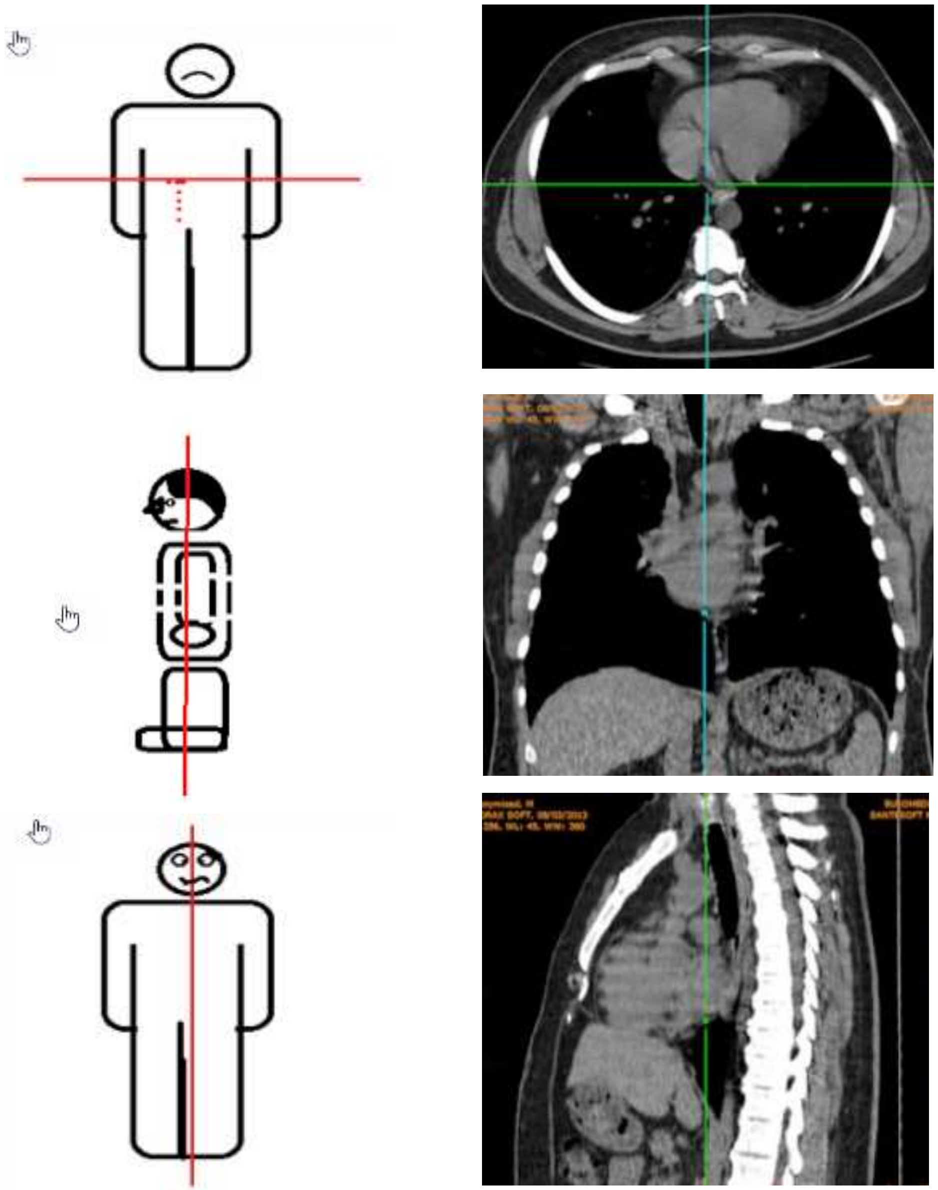


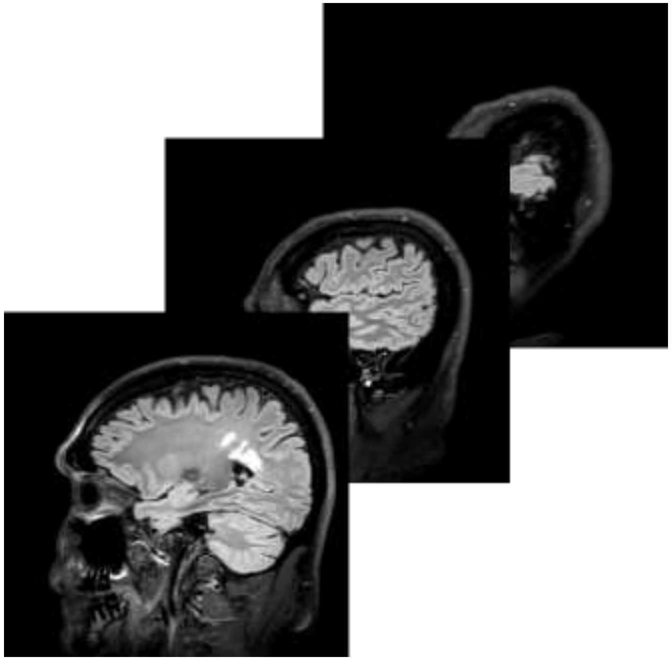
Image IOD
SOP Class UID
SOP Instance UID
SOP Common
Patient
Patients' Name
Patient ID
Patients' Birth Date
Patient Sex
Patient
Study
Study UID
Study Date
Study Time'
Study ID
Referring Physician
Accession Number
General Study
Information Entity
Series
Series UID
Series Number
Modality Type
General Series
Equipment
Manufacturer
Institution Name
General Equipment
Image
System Depended
Acquisition Attributes ...
Position Attributes ...
Image Number
Image Type
General Image
Bits Allocated, Bits Stored
High Bit
Rows, Columns
Samples per Pixel
Planar Configuration
Pixel Representation
Photometric Interpretation
Pixel Data
Image Pixel
Attribute
Module
Window Width
Window Center
VOI LUT

2D Imaging



- **Axial/Transverse**
- **Coronal**
- **Sagittal**

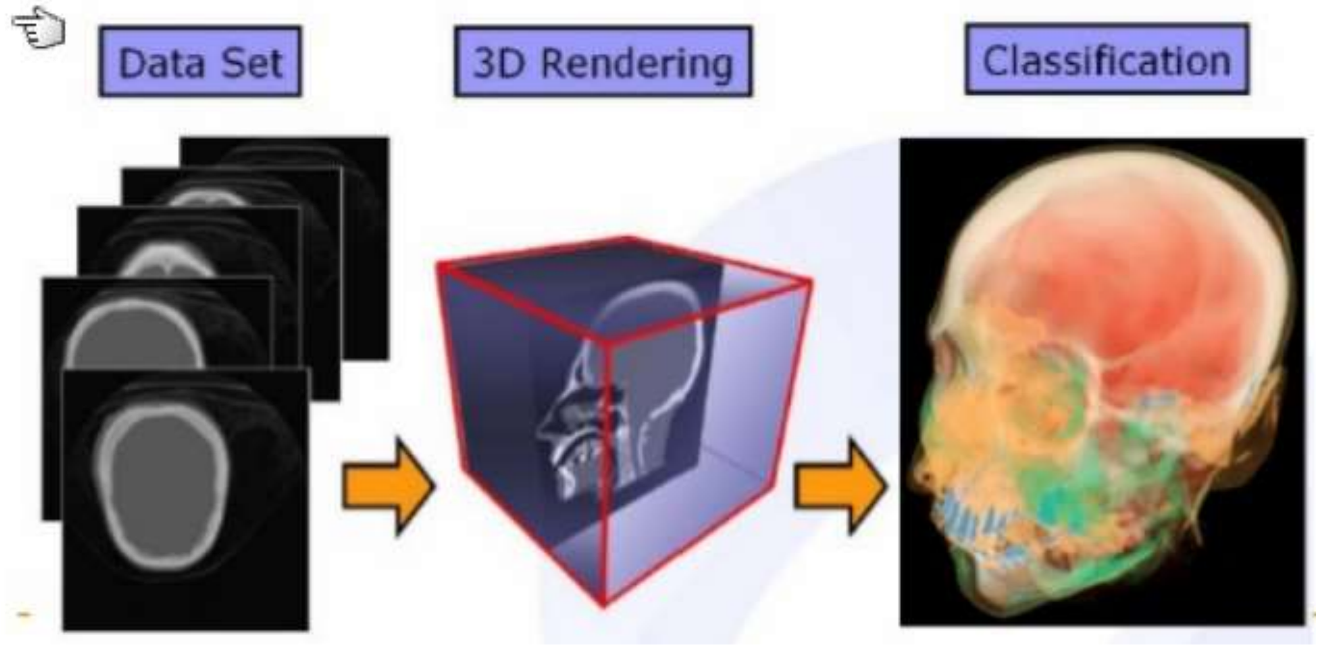
2D Viewing



- **Cine –** Images are viewed in their original plane (say axial) one-by-one
- **MPR (Multi Planar Reformat) –** Volume is resampled along different planes
- **MPR Slab** – Multiple MPR sections are combined/blended
- **Window/Level –** Intensity range of interest is mapped to viewable range of display (usually 8-bit grayscale or 24-bit color)

3D Volume Visualization

- 2D Images are combined to constitute a 3D volume
- 3D Volume is resampled using trilinear sampling to generate:
  - Resampled slices/slabs along any orientation (Obliques / MPR)
  - Isosurfaces
  - Blended projections of the volume (direct volume rendering)
  - Compositing / MIP / MinIP



DICOM Header

IMAGE PLANE MODULE ATTRIBUTES

Attribute Name	Tag	Type	Attribute Description
Pixel Spacing	(0028,0030)	1	Physical distance in the patient between the center of each pixel, specified by a numeric pair - adjacent row spacing (delimiter) adjacent column spacing in mm. See 10.7.1.3 for further explanation.
Image Orientation (Patient)	(0020,0037)	1	The direction cosines of the first row and the first column with respect to the patient. See C.7.6.2.1.1 for further explanation.
Image Position (Patient)	(0020,0032)	1	The x, y, and z coordinates of the upper left hand corner (center of the first voxel transmitted) of the image, in mm. See C.7.6.2.1.1 for further explanation.
Slice Thickness	(0018,0050)	2	Nominal slice thickness, in mm.
Slice Location	(0020,1041)	3	Relative position of the image plane expressed in mm. C.7.6.2.1.2 for further explanation.

Daikon Reader

```
daikon.Dictionary.dict = {
    "0002" : {
        "0001" : ["OB", "FileMetaInformationVersion"],
        "0002" : ["UI", "MediaStoredSOPClassUID"],
        "0003" : ["UI", "MediaStoredSOPInstanceUID"],
        "0010" : ["UI", "TransferSyntaxUID"],
        "0012" : ["UI", "ImplementationClassUID"],
        "0013" : ["SH", "ImplementationVersionName"],
        "0016" : ["AE", "SourceApplicationEntityTitle"],
        "0100" : ["UI", "PrivateInformationCreatorUID"],
        "0102" : ["OB", "PrivateInformation"]
    },
    "0004" : {
```

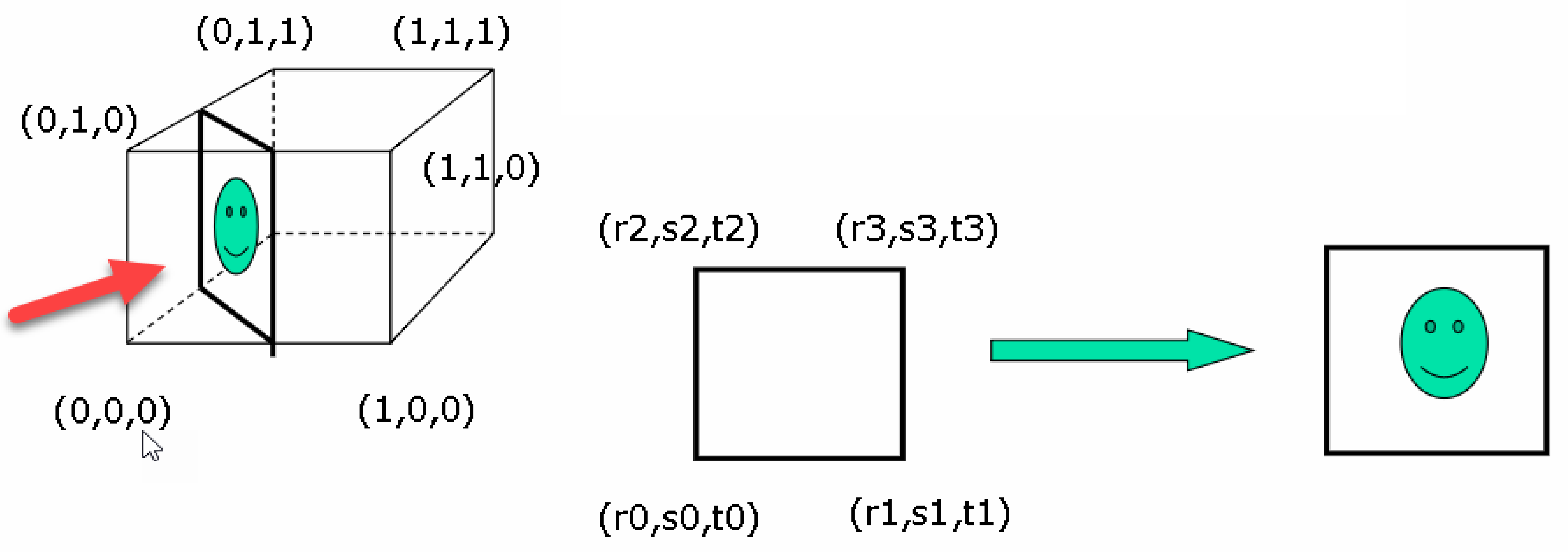
```
daikon.Image.prototype.getImagePosition = function () {
    return daikon.Image.getValueSafely(this.getTag(daikon.Tag.TAG_IMAGE_POSITION[0], daikon.Tag.TAG_IMAGE_POSITION[1]));
};

/**
 * Returns the image axis directions
 * @return {number[]}
 */
daikon.Image.prototype.getImageDirections = function () {
    return daikon.Image.getValueSafely(this.getTag(daikon.Tag.TAG_IMAGE_ORIENTATION[0], daikon.Tag.TAG_IMAGE_ORIENTATION[1]))
};
/**
 * Returns the image position value by index.
 * @param {number} sliceDir - the index
 * @returns {number}
 */
daikon.Image.prototype.getImagePositionSliceDir = function (sliceDir) {
    var imagePos = daikon.Image.getValueSafely(this.getTag(daikon.Tag.TAG_IMAGE_POSITION[0], daikon.Tag.TAG_IMAGE_POSITION[1]));
    if (imagePos) {
        if (sliceDir >= 0) {
            return imagePos[sliceDir];
        }
```

3D Texture Slicing

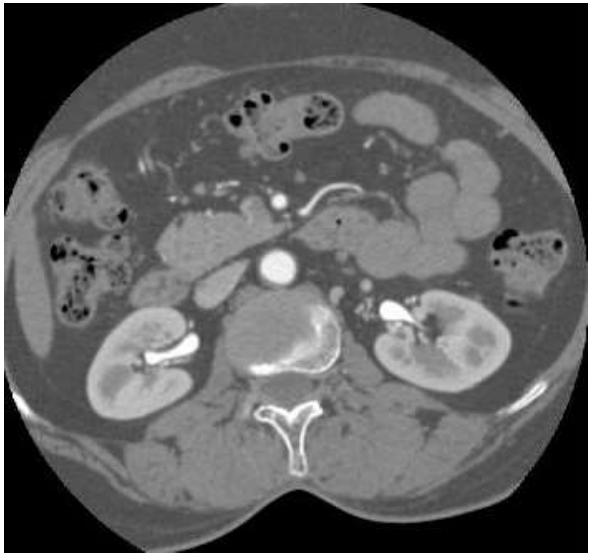
- View-Aligned slices through 3D Texture
- Color values at samples obtained from LUT / Transfer Function
- Slices are blended using Compositing/MIP/MinIP operators

3D Texture Slicing

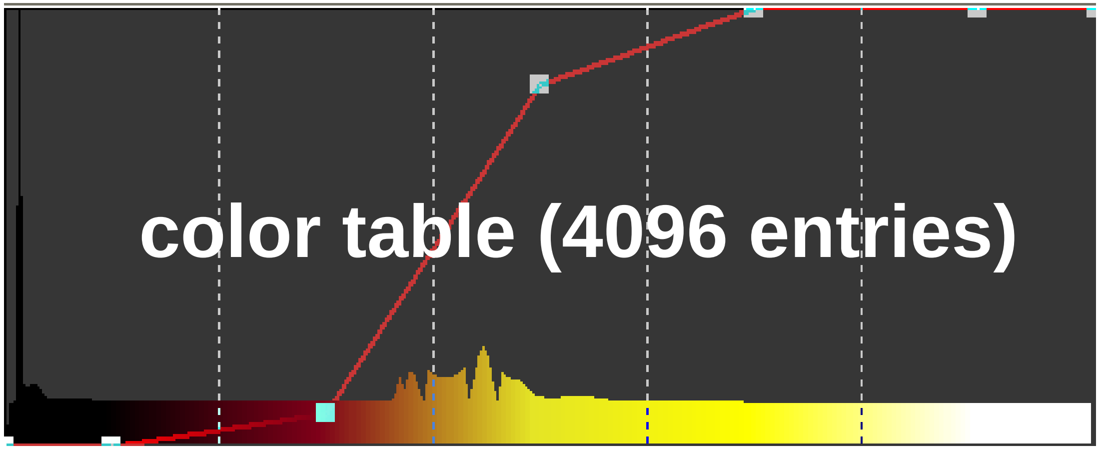
- Transform texture coordinates of slicing quad according to the current view
- Draw textured quad using transformed texture coordinates
- Repeat by stepping the textured quad along the volume extents



Color Tables

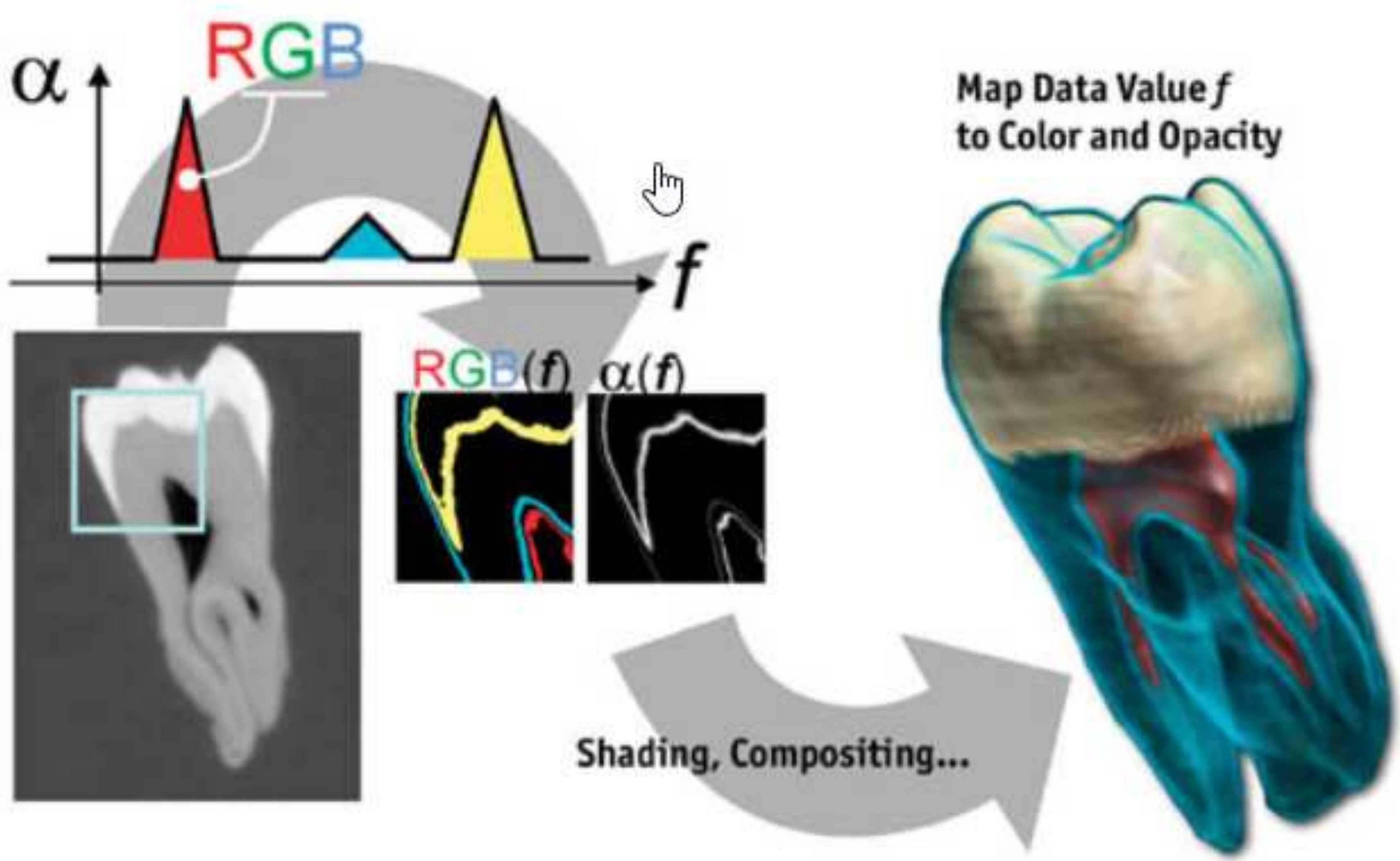


Source image
(12 bit grayscale)



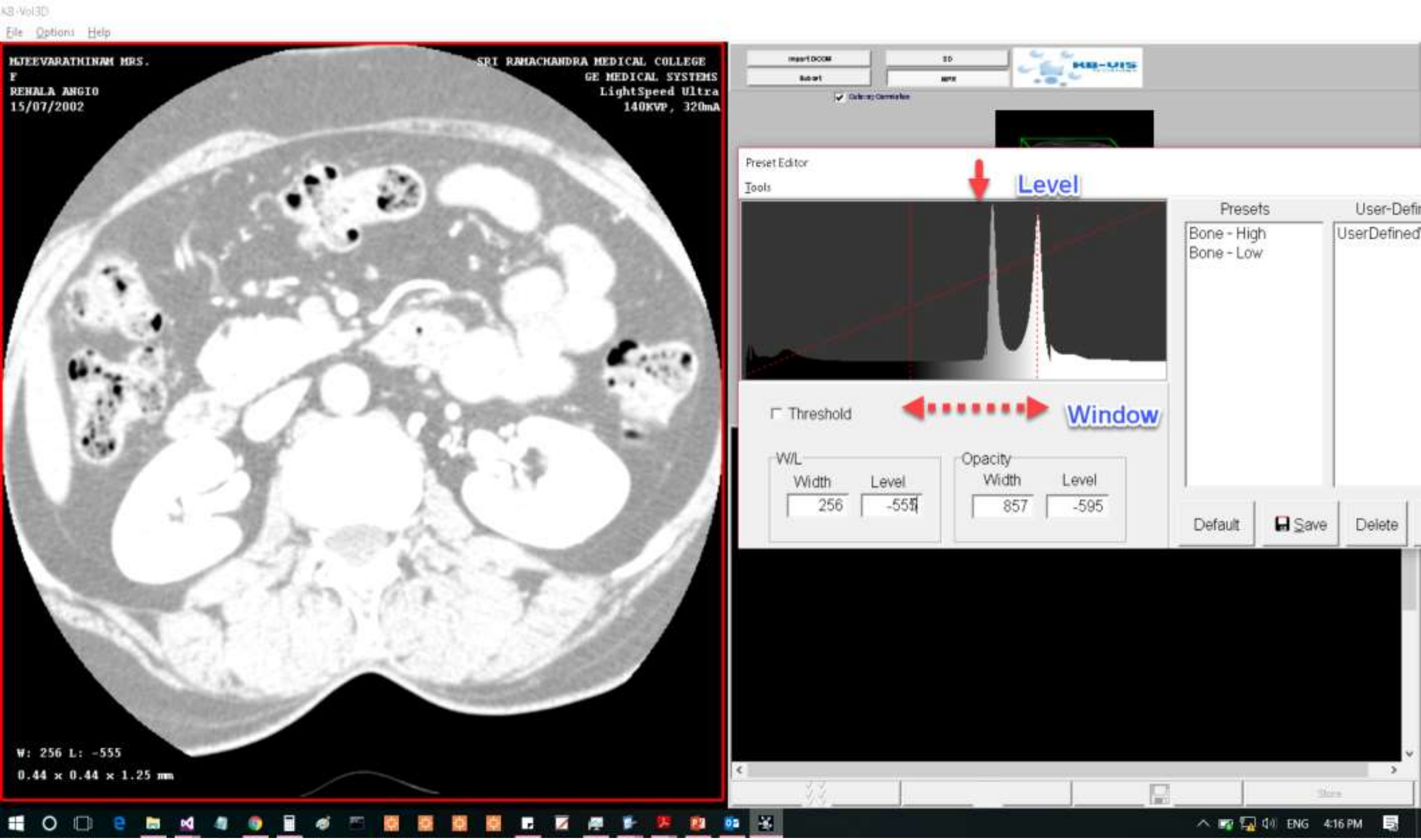
color index lookup:
`(unsigned int)` ➡ `(R,G,B,A)`

Color/Opacity Lookup (Classification)

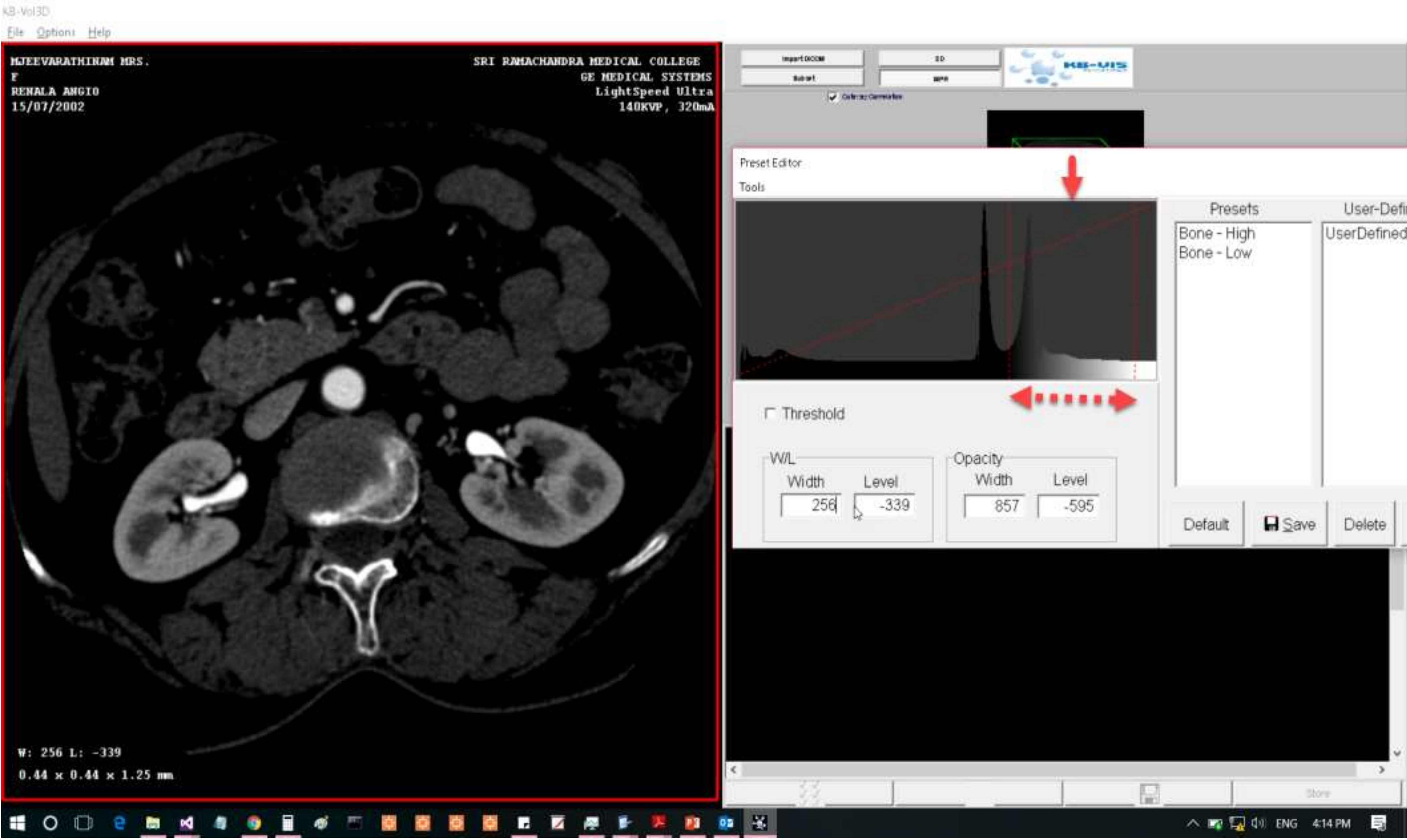


Window/Level

- Select range of intensities to display
- e.g. if DICOM data is 12-bit, then map a range of the 4096-level intensity range to 256-level display range
- Window Width – size of intensity range
- Level – Centre of Window



- Window = 256, Level = -555 (HU)
- Most data visible
- High-intensity areas are saturated



- Window = 256, Level = -339 (HU)
- Less saturation in high-intensity regions
- Low-intensity regions less visible

Review:

Transparency and Blending
Volume Rendering
WebGL 3D Texture

Transparency & Blending

- **Blending** colors to make objects appear translucent

`glEnable(GL_BLEND)`

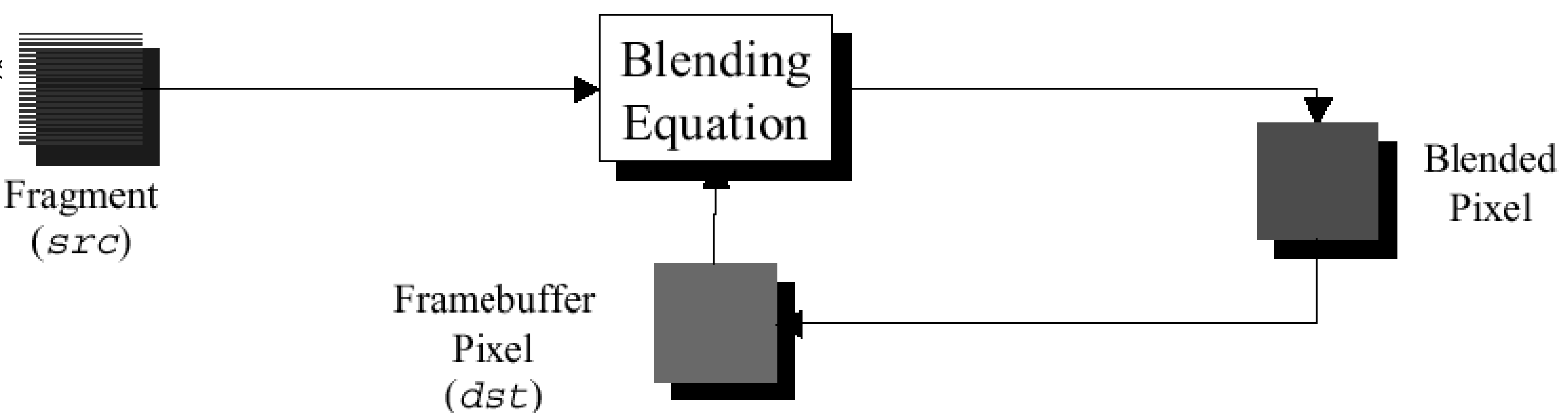
- **Blending function** specifies how color values from a

source and a destination are combined:

`glBlendFunc(GLenum sfactor, GLenum dfactor)`

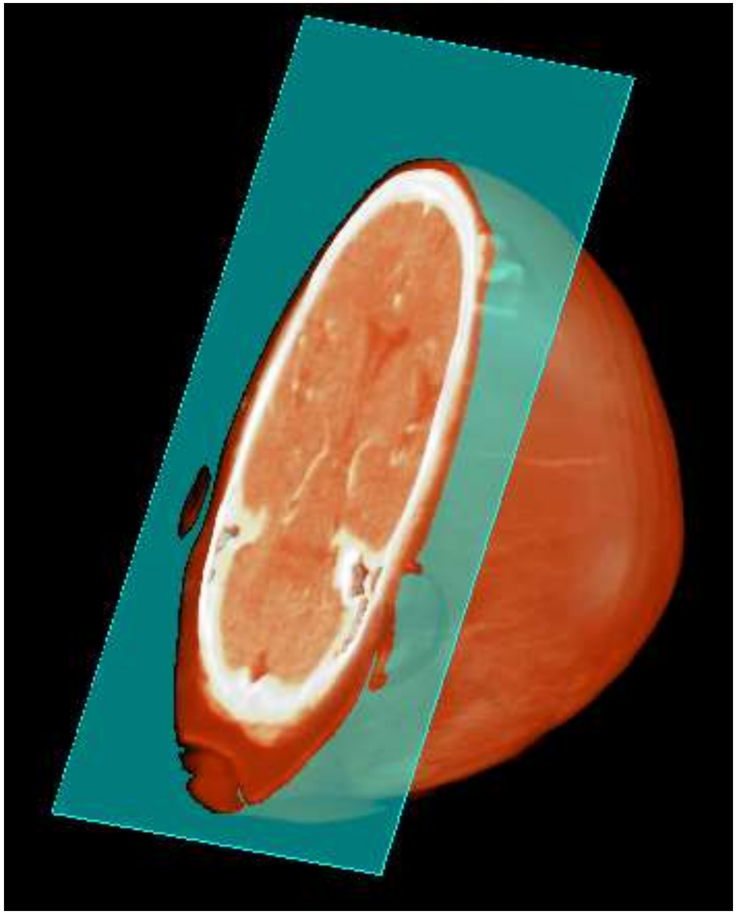
- color values of incoming fragment (*source*) are combined with the color values of the corresponding currently stored pixel (*destination*):

Cde



Blend Function

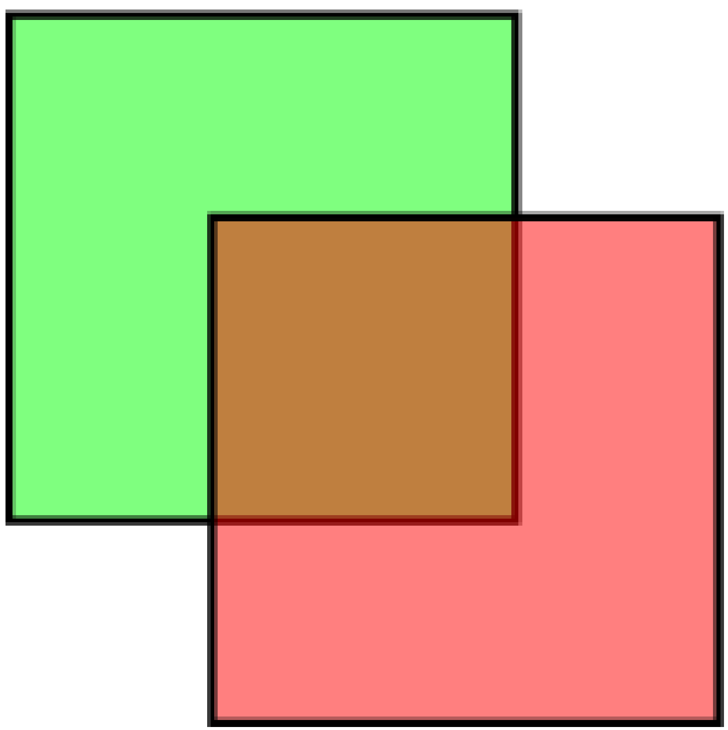
`glBlendEquation(GL_MAX)`



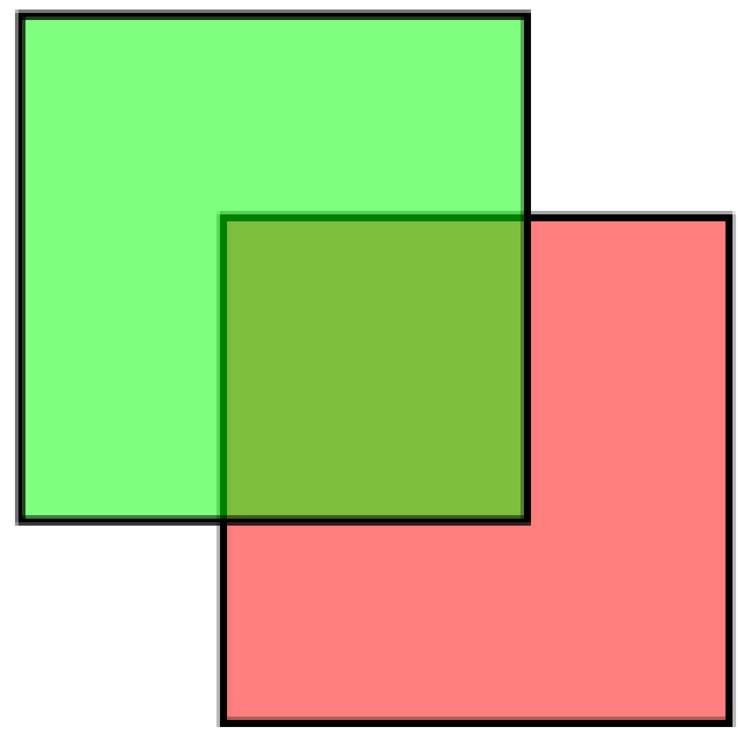
`glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`

“Over” Operator

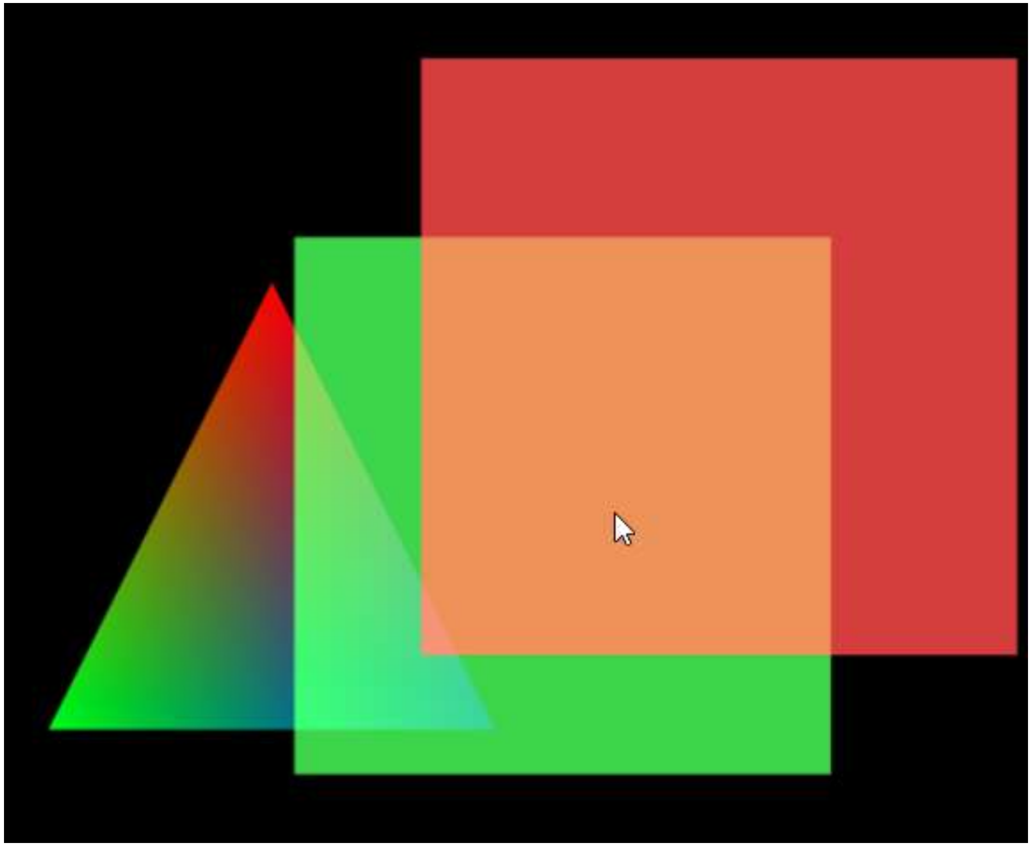
Red on top



Green on top

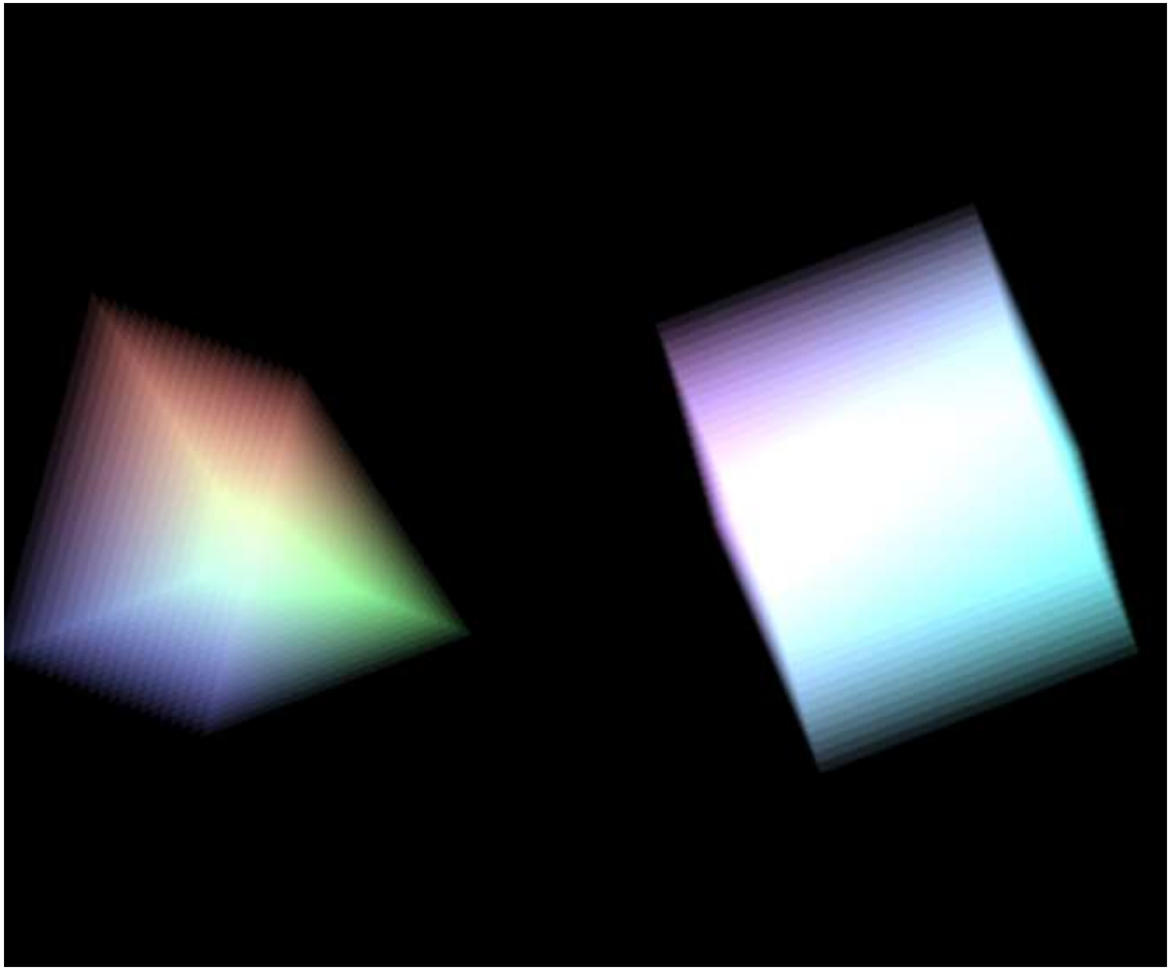


Example: Blending



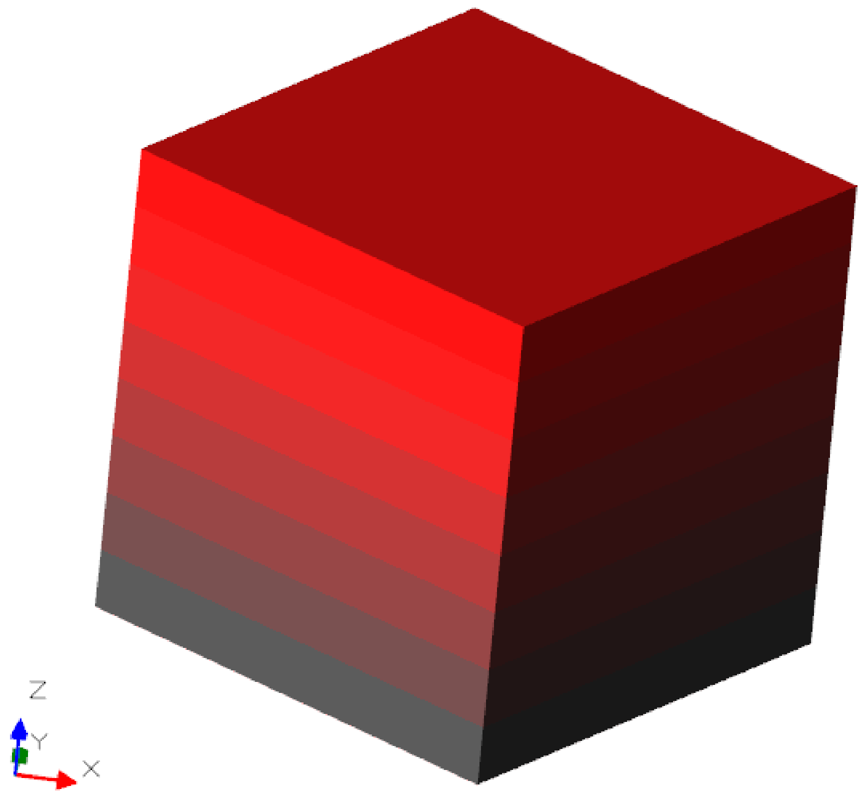
See: example10-transparency-and-blending

Example: Transparent Stack



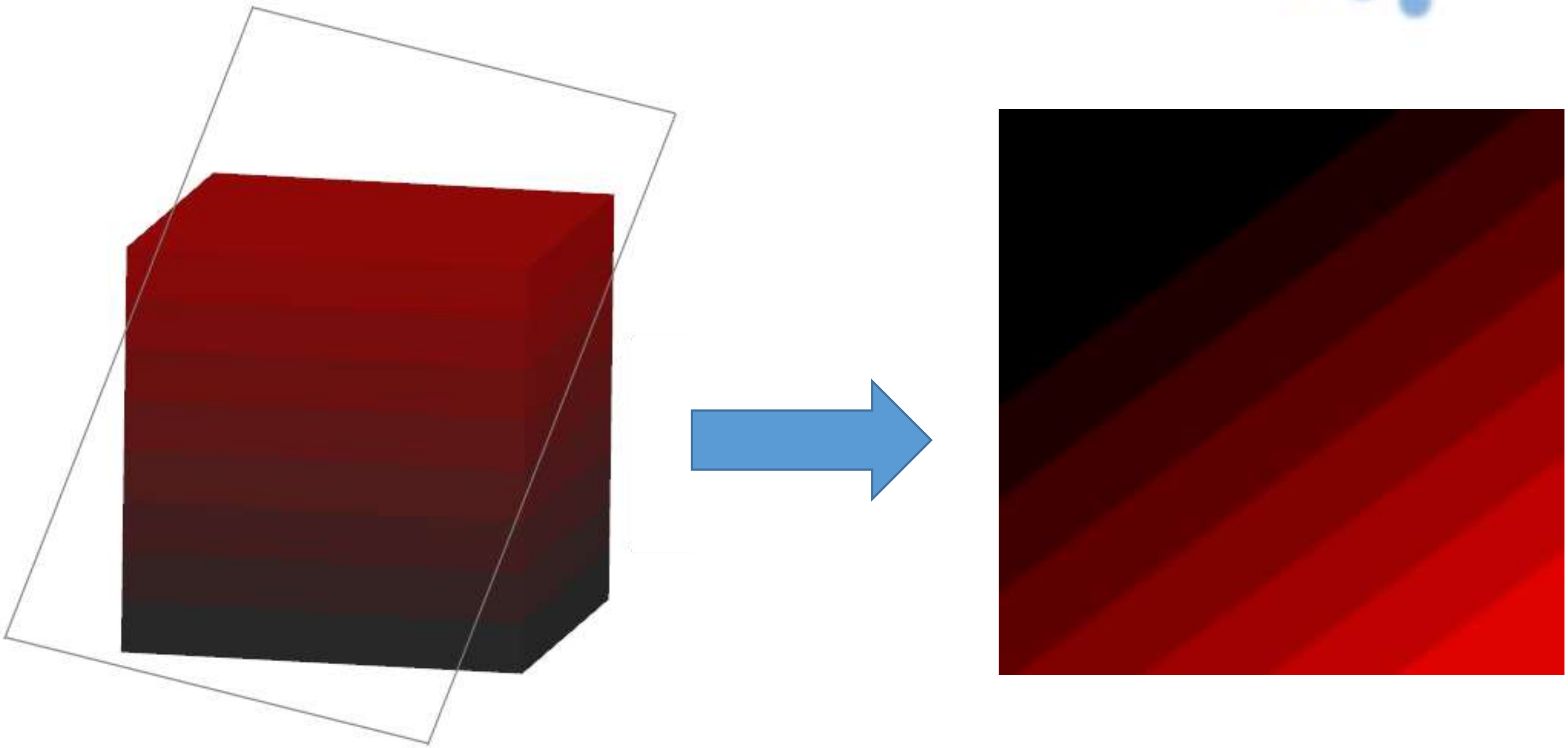
See: example10-transparency-and-blending

Example: 3D Texture



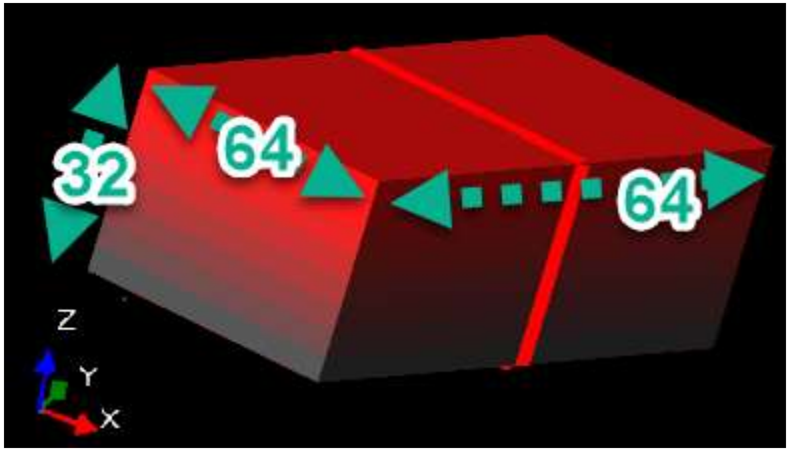
See: example11-3d-texture

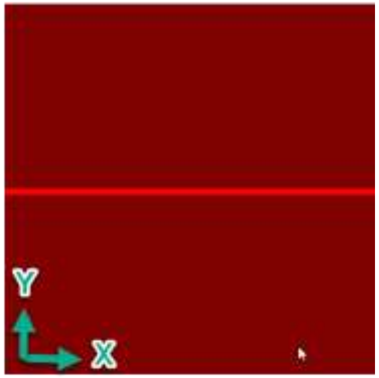
Sampling the 3D Texture



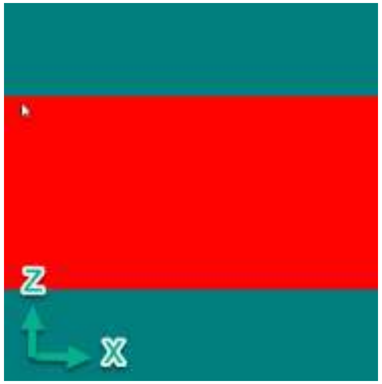
- Use 3D texture coordinates (s,t,p) as vertex attributes
- In **glTexParameter** use TEXTURE_WRAP_R for third texture dimension

Simplified Example – Standard Slice Planes

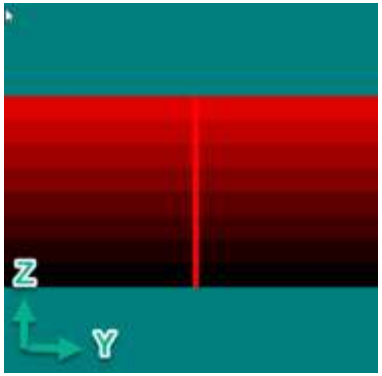




Axial (XY)



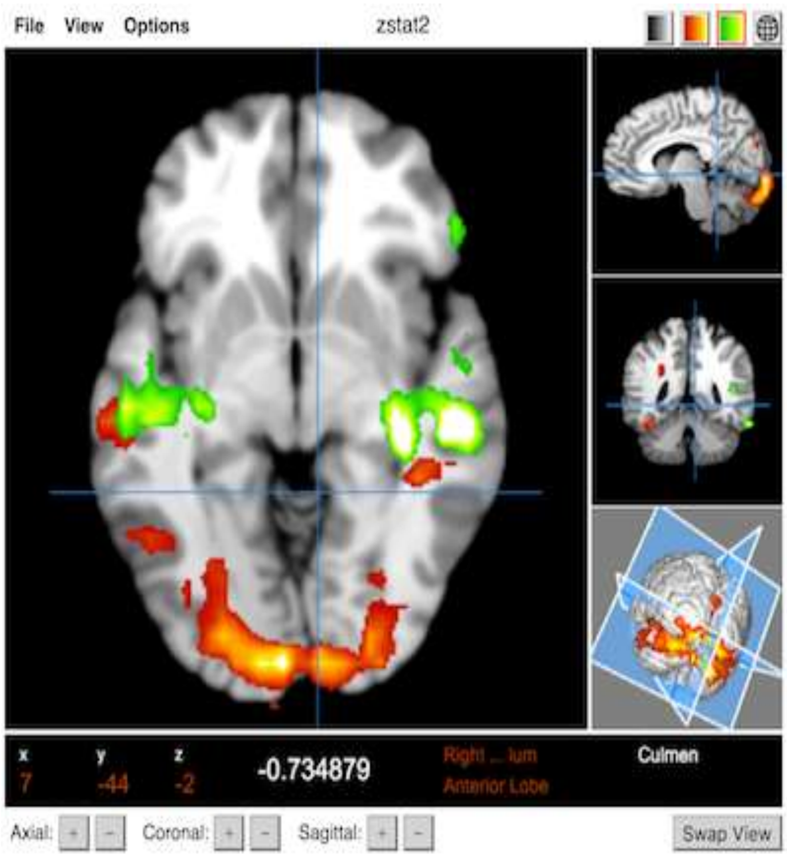
Coronal (XZ)



Sagittal (YZ)

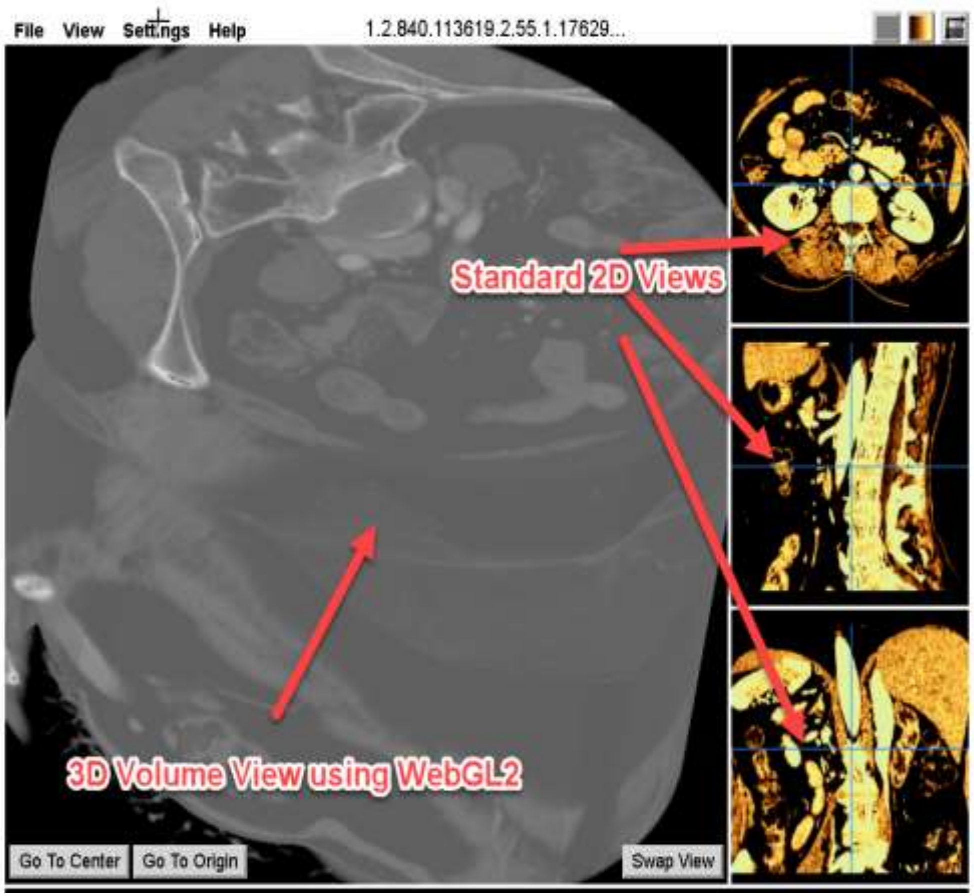
DAY 2

Papaya Viewer



- Open-source Javascript DICOM viewer
- Uses Daikon DICOM reader
- 2D Standard Views – no obliques or slabs
- Isosurfacing
- No Volume Rendering
- Uses Javascript – no graphics acceleration
- Slow on large data

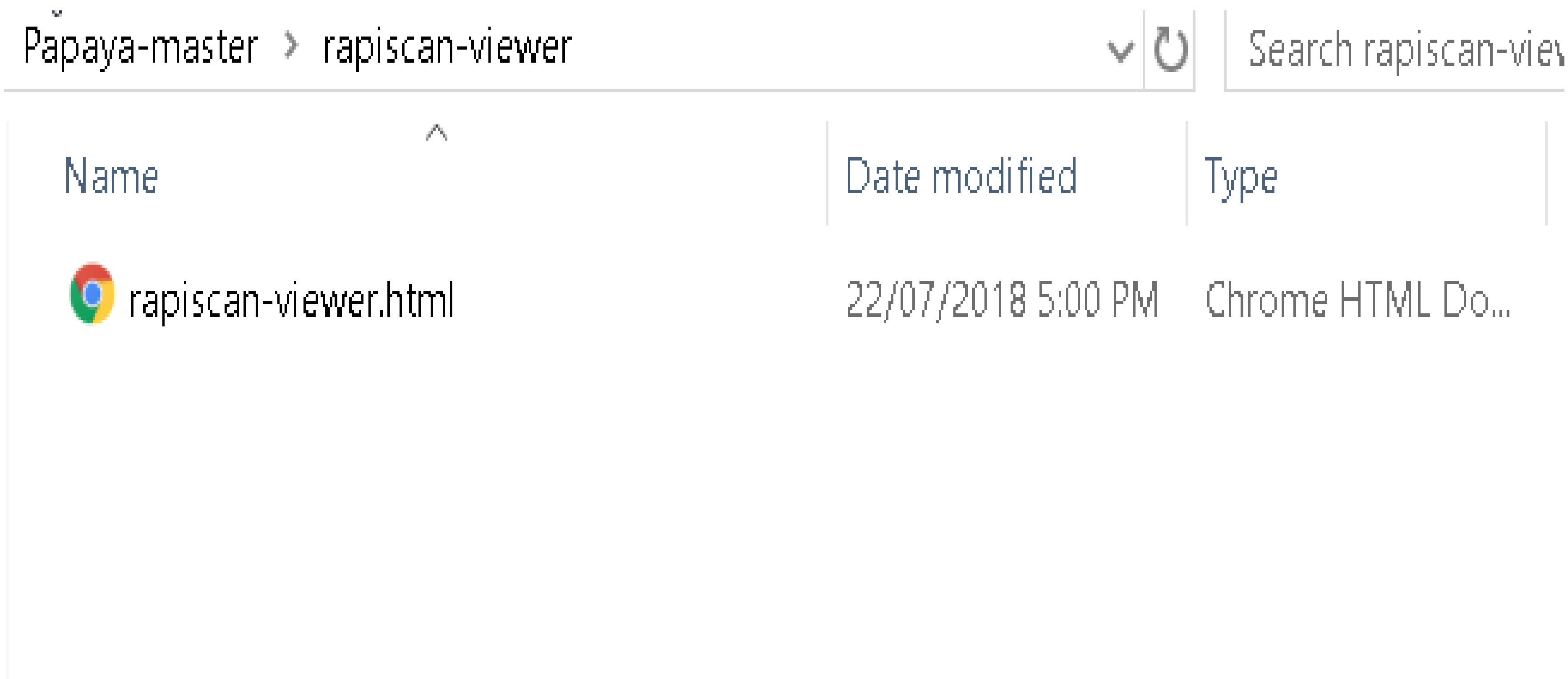
Rapiscan WebGL Volume Viewer



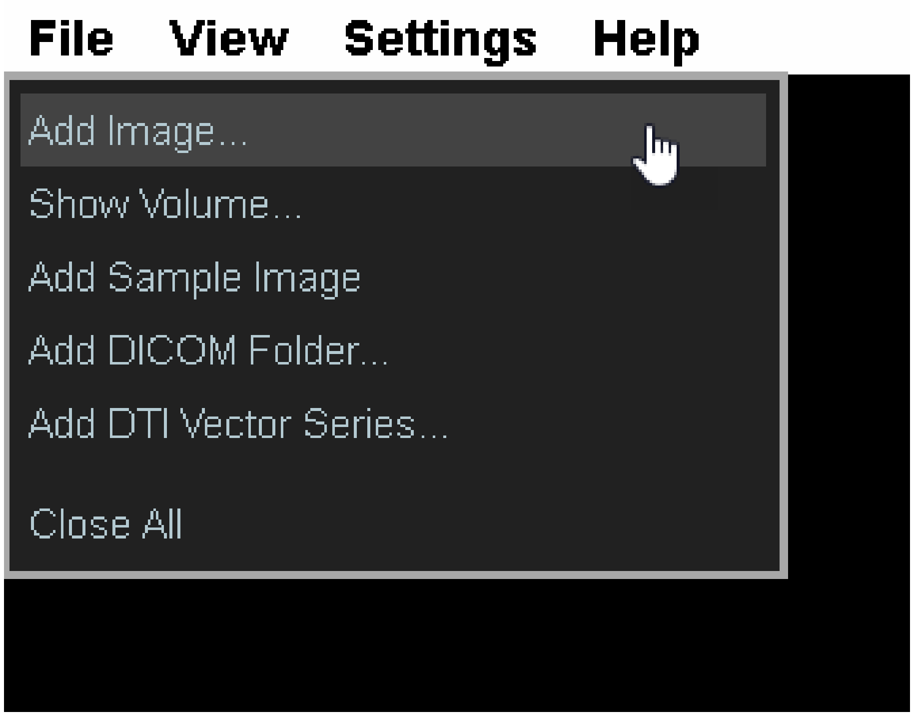
- 3D Volume Rendering view based on WebGL 2.0
- Volume rendering code in rapiscanVolume.js added to Papaya codebase
- Grayscale, Color, and Transparency with Window/Level and Opacity control
- Integrated with Papaya Color Tables
- Cut-Planes dynamically updating with MPR crosshairs

Source Code: rapiscan-examples-webgl\rapiscan-papaya-viewer

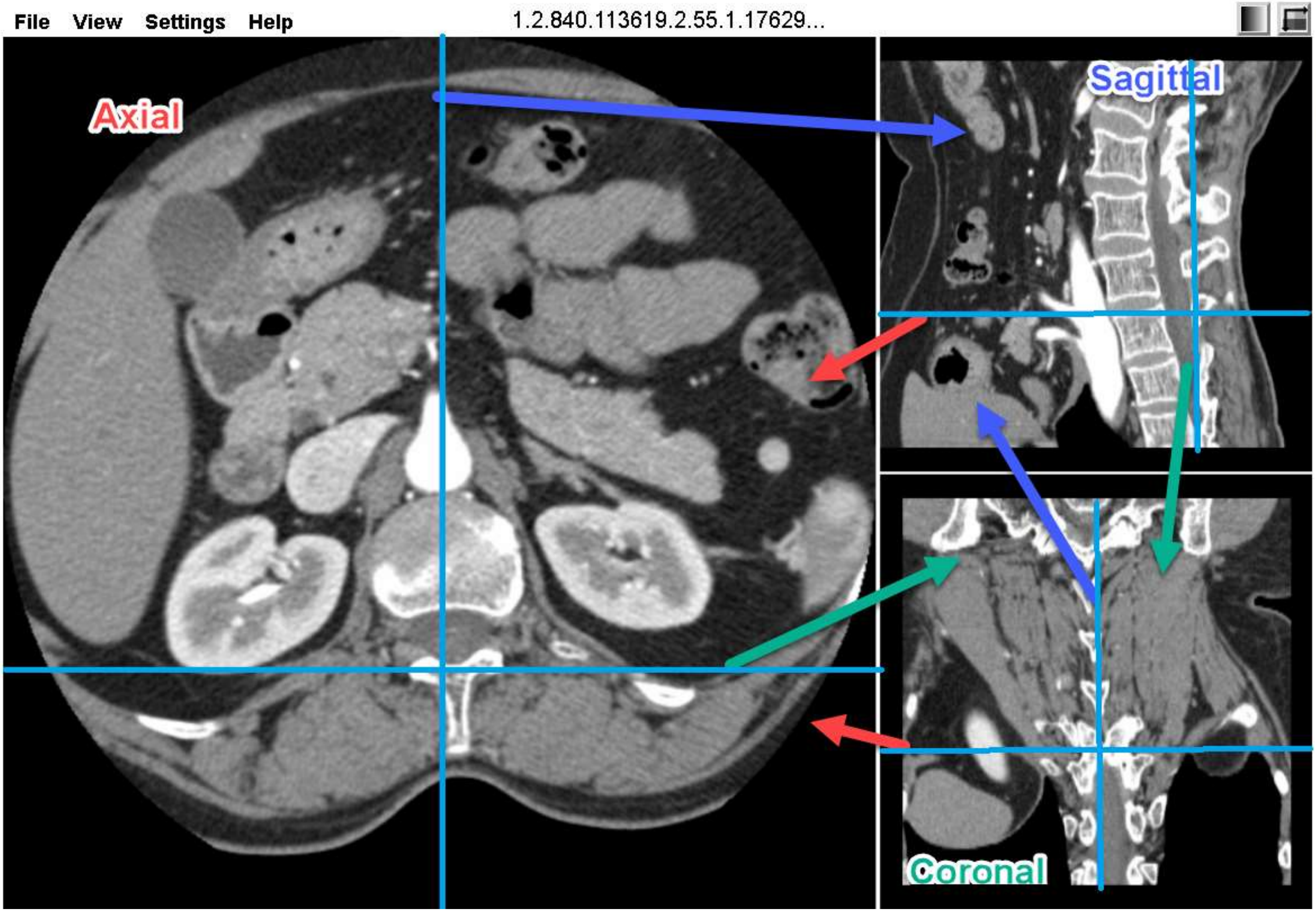
Rapiscan WebGL Viewer



Load DICOM Series

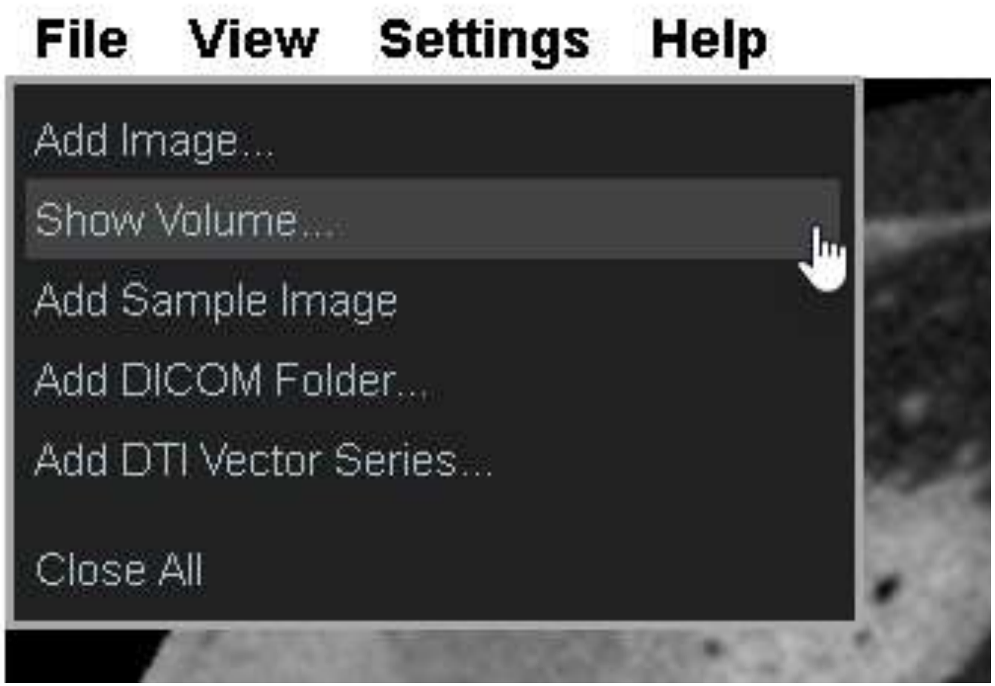


- Use Add Image, and select a .DCM multiframe volume series file
- Or, use Add DICOM Folder and choose folder containing files from a single Volume Series

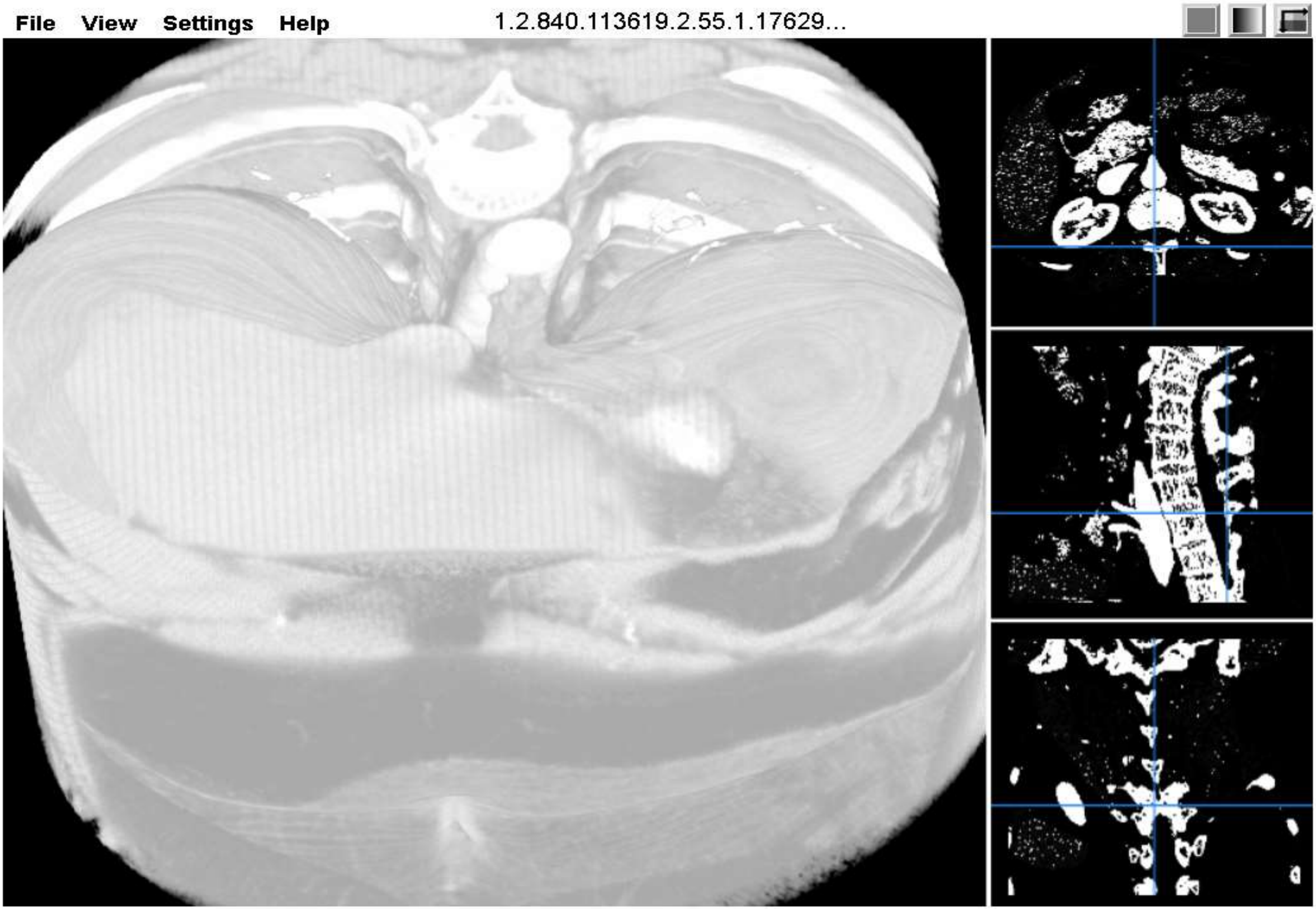


- 2D Views (MPR) – Axial, Coronal Sagittal
- Moving crosshairs updates the MPR views

Load Volume

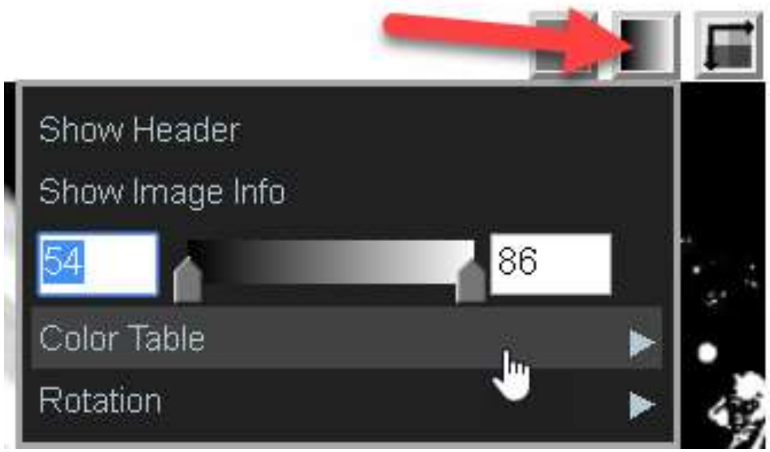


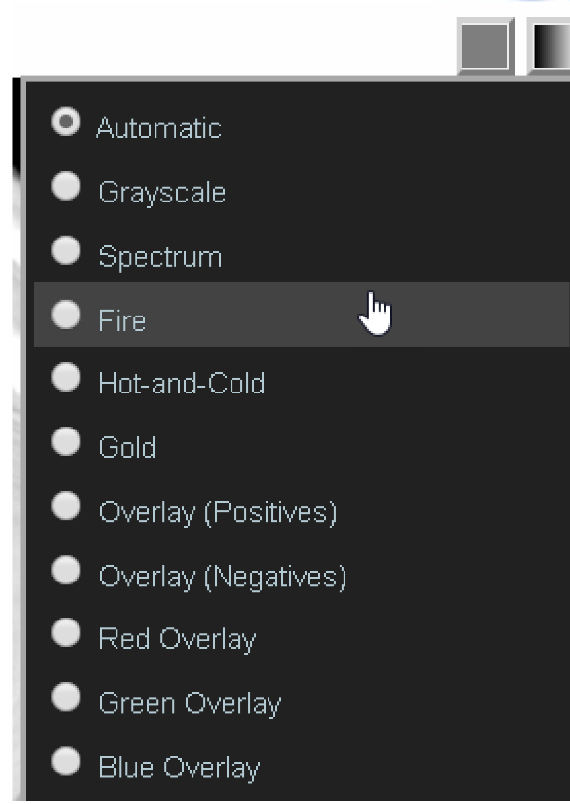
- Loads current series as 3D Volume
- Current series must constitute a valid DICOM volume



- Volume is loaded with default window/level based on data histogram

Color Tables



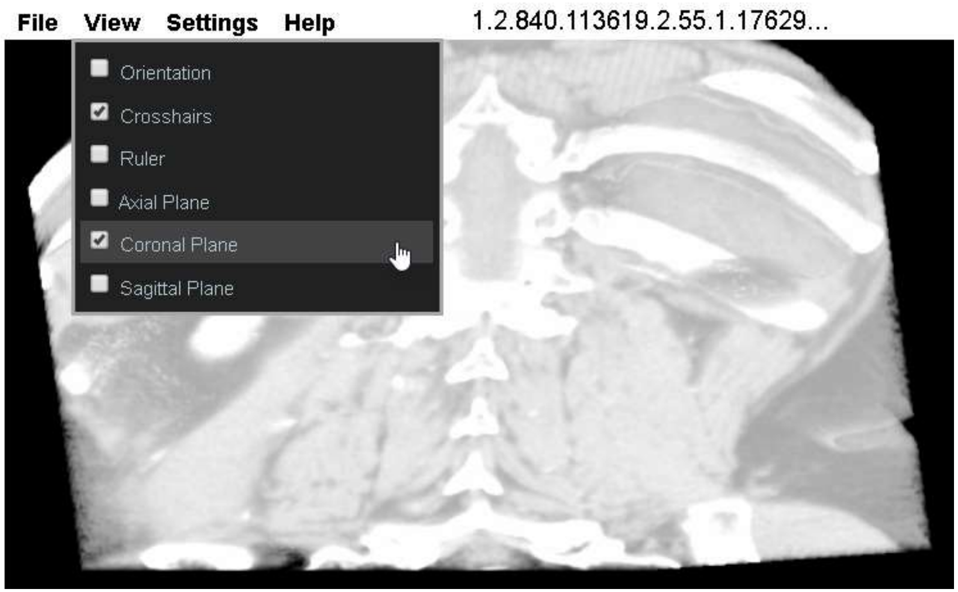


- Grayscale/Color RGBA LUTs
- 16-bit intensity is used to look up 256-entry table

Window-Level and Opacity

- Right-mouse + Horizontal drag ➡ Window width
- Right-mouse + Vertical drag ➡ Level
- Press Ctrl for Opacity

Cut-Planes



- Axial/Coronal/Sagittal Cut-Planes
- Cut-Plane updates when corresponding crosshair is moved

Papaya Viewer Source

Papaya-master > src > js >

Search js

Name	Date modified	Type
core	12/07/2018 10:33 ...	File folder
data	12/07/2018 10:33 ...	File folder
surface	12/07/2018 10:33 ...	File folder
ui	17/08/2018 1:30 A...	File folder
utilities	12/07/2018 10:33 ...	File folder
viewer	18/08/2018 2:29 PM	File folder
volume	12/07/2018 10:33 ...	File folder
constants.js	12/07/2018 10:33 ...	JavaScript File
main.js	17/08/2018 1:29 A...	JavaScript File

Rapiscan WebGL Viewer Source

Papaya-master › src › js › viewer

Search viewer

Name	Date modified	Type
atlas.js	12/07/2018 10:33 ...	JavaScript File
colortable.js	17/08/2018 9:41 PM	JavaScript File
display.js	12/07/2018 10:33 ...	JavaScript File
preferences.js	17/08/2018 1:33 A...	JavaScript File
rapiscanvolume.js	18/08/2018 2:29 PM	JavaScript File
screenslice.js	15/07/2018 12:21 ...	JavaScript File
screensurface.js	22/07/2018 3:44 PM	JavaScript File
screenvol.js	16/08/2018 4:38 PM	JavaScript File
viewer.js	17/08/2018 1:29 A...	JavaScript File

Main Components

- **Main**.js – application container
- **Toolbar**.js – UI Commands, Settings
- **Viewer**.js – main viewer class
- **ScreenVol**.js – encapsulates DICOM volume
- **ScreenSlice**.js – 2D MPR viewer
- **RapiscanVolume.js – WebGL2 Volume Viewer**
- **ColorTable**.js – RGBA Color Tables
- **Daikon**.js – DICOM parsing

papaya.viewer.Viewer Key Members

- **screenVolumes –** volume containers currently loaded. A screen volume contains a DICOM volume and its associated properties and settings e.g. color table
- **currentScreenVolume –** screen volume currently active in viewer
- **volume –** base DICOM volume loaded in viewer (any additional volumes are overlaid on this). This is used by the volume viewer
- **axial/coronal/sagittalSlice** – ScreenSlice instances containing MPR 2D views
- **volumeView** – Volume rendered view
- **mainImage/lowerImageBot/lowerImageTop/ lowerImageBot2** – views mapped to volume and MPR views (can be swapped)
- **currentCoord** – Current MPR crosshair position
- **selectedSlice** – Currently active slice, that user action is originating from

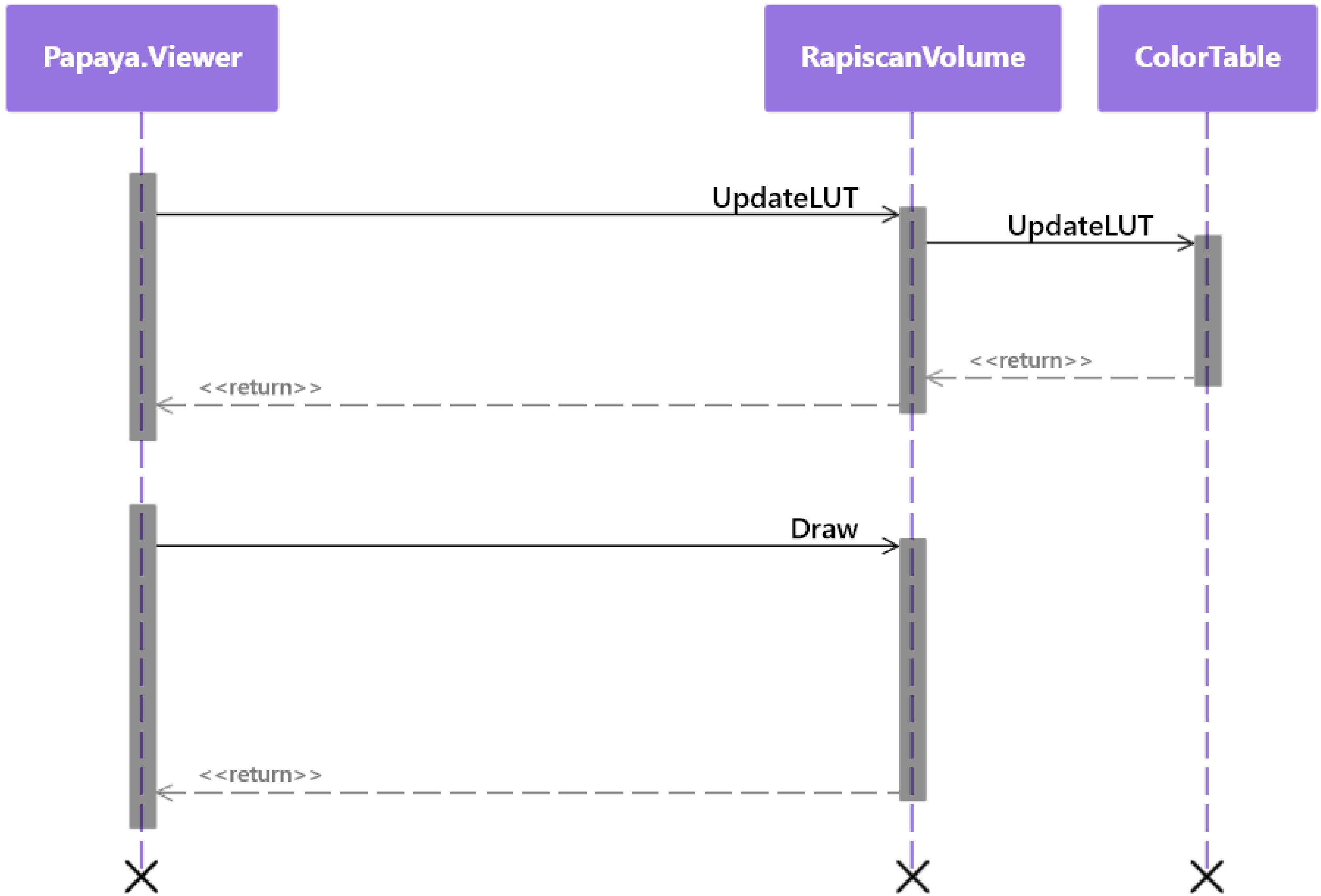
papaya.viewer.Viewer Key Methods

- **drawViewer** – top-level draw, triggers all other necessary draws
- **drawScreenSlice –** recomputes and draws 2D MPR views. Implemented in Javascript, does not use WebGL
- **drawCrosshairs** – draws MPR crosshairs based on current cursor position (currentCoord)
- **windowLevelChanged** – updates W/L (e.g. on mouse drag) and redraws slices, passes on event to volume view
- **opacityChanged** - passes on event to volume view
- **load3dVolume** – creates volume screen container and adds volume view
- **initializeVolume** – creates RapiscanVolume instance with currently loaded DICOM volume
- **updatePosition** – updates current position in volume, updates MPR views, triggers volume update if cut-plane is enabled
- **mouse event handlers –** trigger update of 2D and volume views e.g. W/L, rotate, zoom, crosshair drag

papaya.viewer.RapiscanVolume Interface

- Constructor:
  - **papaya.viewer.RapiscanVolume**()
- Volume view refresh:
  - **draw**()
- Select color table (Settings)
  - **changeColorTable**(lutName)
- Window/Level and Opacity mouse action:
  - **updateLut**(minIntensity, maxIntensity, isOpacity)
- Cut-plane update on MPR crosshair drag:
  - **updateVolume**(currentCoord, draggingSliceDir)

RapiscanVolume Events



RapiscanVolume Methods

Initialize()

- Initializes canvas, WebGL2 context
- Initializes the view (orthographic projection)
- Sets up shaders
- Creates and initializes GL buffers

RapiscanVolume Methods

Draw()

- Calls **Initialize()** the first time
- Calls **DrawScene()** for GL drawing

DrawScene()

- Sets up matrices based on current rotation
  - Updates **mvMatrix** based on rotation state
  - Updates **texMatrix** (used to transform texture coordinates of the volume sampling/slicing plane)
- Sets shader uniforms
- Calls **BindVolume()** to load the volume (initially)
- Calls **RenderVolume()** to render the volume

RapiscanVolume Methods

BindVolume()

- creates 3D texture and load its using the 8/16-bit volume data from the DICOM file
- 3D texture is loaded using **glTexImage3D()**
- 16-bit intensities are stored in UNSIGNED_SHORT_4_4_4_4 texture format and repacked in the shader
- ~~3D texture is padded to power-of-2 dimensions, as WebGL support for NPOT textures is patchy~~ *(no longer needed)*

RenderVolume()

- draws volume rendered view
- draws stack of view-aligned slices that are 3D textured using the volume data
- blending is enabled, and fragments are composited using the "over" operator

RenderVolume()

// draw a stack of view-aligned slices that are 3D textured using the volume data

```
// generate enough slices to cover volume extent
    var sliceCount = this.maxDim;

// use matrix to transform texture coordinates of textured quad
    var tempMat = mat4.create();
    mat4.set(this.texMatrix, tempMat);  // texMatrix contains current volume rotation
    var zdir = vec3.create();
    vec3.set([0,0,1], zdir);  // view direction is along z-axis
    var startTrans = vec3.create();
    vec3.scale(zdir, -1.732/2.0, startTrans); // translate to one end of the volume
    mat4.translate(tempMat, startTrans);
    var viewStep = vec3.create();
    vec3.scale(zdir, 1.732 / sliceCount, viewStep); // compute incremental step vector
```

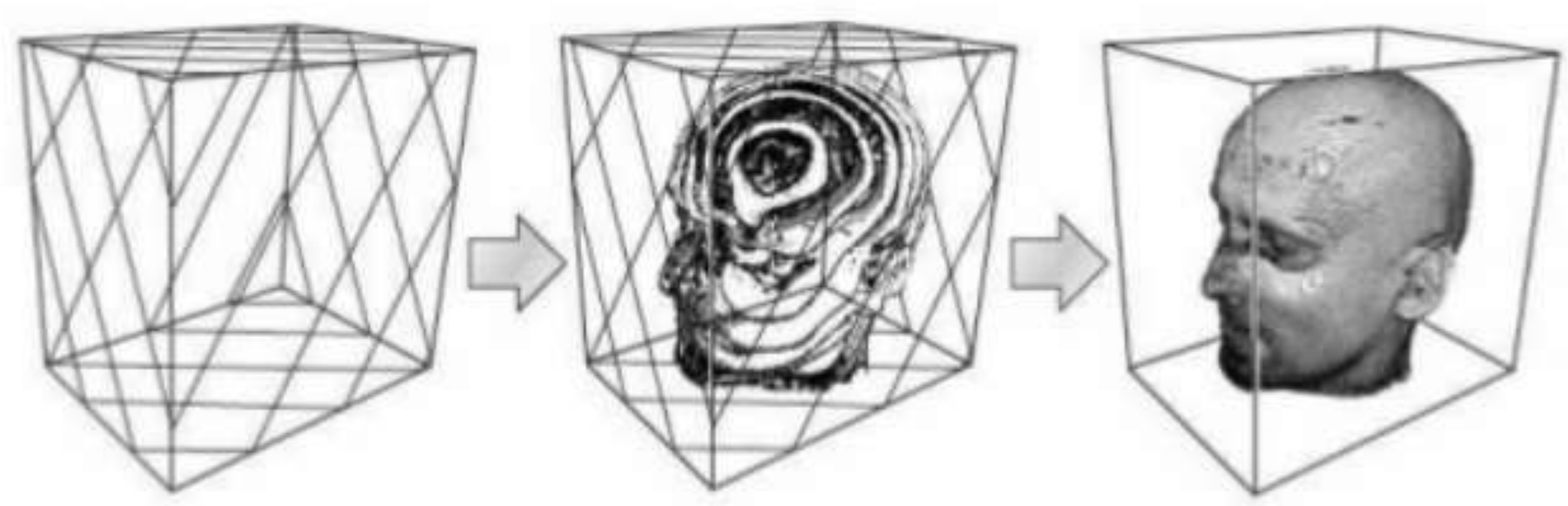
RenderVolume()

// repeatedly draw textured slice, stepping through the volume,
// with blending enabled

```
for (var slice = 0; slice < sliceCount; ++slice) {
    // step through the stack by applying incremental translation
    mat4.translate(tempMat, viewStep);
    // pass updated matrix to shader
    gl.uniformMatrix4fv(this.shaderProgram.texMatrixUniform, false,
        tempMat);
    // draw the textured quad (view-aligned volume slice)
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

3D Texture Slicing

- View-Aligned slices through 3D Texture
- Color values at samples obtained from LUT / Transfer Function
- Slices are blended using Compositing/MIP/MinIP operators



Vertex Shader

```
in vec3 aVertexPosition;
in vec3 aVertexTexCoord;

 // projection matrix
uniform mat4 uPMatrix;
// transforms texture coordinates of textured slice
uniform mat4 uTexMatrix;

// output to fragment shader
out vec3 vTransformedTexCoord;

void main(void) {
     // transform texture coordinates to account for volume rotation
     // and current slice position
    vec4 transformedTexCoord = uTexMatrix * vec4(aVertexTexCoord, 1.0);
     // pass to fragment shader for interpolation
    vTransformedTexCoord = transformedTexCoord.xyz;

     gl_Position = uPMatrix * vec4(aVertexPosition, 1.0);
 }
```

Fragment Shader

```
uniform vec3 uVolumeCoord; // current slice plane location
uniform bool uVolumeSlice; // in volume render mode?
uniform int uSlicePlane; // slice plane linked to axial / coronal / sagittal MPR
uniform int uBytesPerPixel; // 1 or 2 (8 or 16 bits)

uniform sampler3D uVolumeSampler; // volume texture
uniform sampler2D uLutSampler; // lookup table texture (256x1 2D texture)

in vec3 vTransformedTexCoord;

out vec4 fragColor;

void main(void) {

   fragColor = vec4(0.0,0.0,0.0,1.0);

   // discard fragment if outside texture limits
   if(vTransformedTexCoord.x < 0.0 || vTransformedTexCoord.y < 0.0
                 ||vTransformedTexCoord.z < 0.0 || vTransformedTexCoord.x > 1.0
        || vTransformedTexCoord.y > 1.0 || vTransformedTexCoord.z > 1.0){
                 discard;
   }
```

Fragment Shader

```
if (uVolumeSlice) {
   // sample 3D texture along the current slice
   fragColor = texture(uVolumeSampler, vTransformedTexCoord);
   // repack 4-bit components into single 16-bit unsigned int
   float lutIndex = uBytesPerPixel > 1 ?
         fragColor.g + fragColor.b/16.0 + fragColor.a/256.0 : fragColor.r;
   // look up volume texture using 16-bit intensity index
   fragColor = texture(uLutSampler, vec2(lutIndex, 0.0));
   if(fragColor.a < 0.05) discard; // very low alpha, do not output
   else{
        // if slice plane is enabled, then check if fragment is on visible
           side, or is clipped away
       if(uSlicePlane == 1){   //AXIAL
           if(dot(vTransformedTexCoord - vec3(0.5,0.5,0.5),
                  vec3(0.0,0.0,1.0)) < uVolumeCoord.z - 0.5) discard;
       }else if(uSlicePlane == 2){ //CORONAL
           if(dot(vTransformedTexCoord - vec3(0.5,0.5,0.5),
                  vec3(0.0,1.0,0.0)) < uVolumeCoord.y - 0.5) discard;
       }else if(uSlicePlane == 3){ //SAGITTAL
           if(dot(vTransformedTexCoord - vec3(0.5,0.5,0.5),
                  vec3(1.0,0.0,0.0)) < uVolumeCoord.x - 0.5) discard;
       }
   }
}
```

Surface Shading

```
 if(uShaded){
// sample volume on either side of current voxel
vec3 gradient;
	gradient.r = intensity(texture(uVolumeSampler, transformedTexCoord(vVertexTexCoord
+ vec3(uVoxelSize.x, 0.0, 0.0))),uBytesPerPixel);
	gradient.g = intensity(texture(uVolumeSampler, transformedTexCoord(vVertexTexCoord
+ vec3(0.0, uVoxelSize.y, 0.0))),uBytesPerPixel);
	gradient.b = intensity(texture(uVolumeSampler, transformedTexCoord(vVertexTexCoord
+ vec3(0.0, 0.0, uVoxelSize.z))),uBytesPerPixel);
	vec3 gradient_back;
	gradient_back.r = intensity(texture(uVolumeSampler,
transformedTexCoord(vVertexTexCoord - vec3(uVoxelSize.x, 0.0, 0.0))),uBytesPerPixel);
	gradient_back.g = intensity(texture(uVolumeSampler,
transformedTexCoord(vVertexTexCoord - vec3(0.0, uVoxelSize.y, 0.0))),uBytesPerPixel);
	gradient_back.b = intensity(texture(uVolumeSampler,
transformedTexCoord(vVertexTexCoord - vec3(0.0, 0.0, uVoxelSize.z))),uBytesPerPixel);
// compute surface normal using central-difference
	gradient = 0.5*(gradient - gradient_back);
```

Surface Shading

```
if(length(gradient) > 0.05){
        gradient = normalize(gradient);
        // assume light and viewing directions along z-axis
        vec3 lightVec = vec3(0.0,0.0,1.0);
        vec3 viewVec = vec3(0.0,0.0,1.0);
        // compute dot product of normal and light vector
        float NdotL = dot(-lightVec, gradient);
        // compute dot product of normal and light vector
        float NdotH = clamp(dot(viewVec, gradient), 0.0, 1.0);
        // clamp to get diffuse luminance (when NdotL < 0, surface is not light-facing)
        float lum = clamp(NdotL, 0.0, 1.0);
        float gloss = 20.0;
        // compute specular component (for shininess)
        float spec = clamp(pow(NdotH, gloss), 0.0, 0.4);
        // final shaded luminance
        lum = clamp(lum + spec, 0.0, 1.0);
        float brightness=1.0;
        float contrast=1.0;
        fragColor.r = clamp(contrast*(lum*fragColor.r-0.5)+ 0.5*brightness, 0.0, 1.0);
        fragColor.g = clamp(contrast*(lum*fragColor.g-0.5)+ 0.5*brightness, 0.0, 1.0);
        fragColor.b = clamp(contrast*(lum*fragColor.b-0.5)+ 0.5*brightness, 0.0, 1.0);
    }
  }
```

RapiscanVolume Methods

ChangeColorTable()

- Loads new color selected in Papaya viewer
- Calls updateLut() to update the LUT

UpdateLut()

- Updates color table due to window-level/opacity change or color table change
- Populates RGBA LUT based on current min/max intensity range
- Calls **colorTable.lookupRed/Green/Blue()** to get RGB value for a given intensity
- Updates LUT texture using **glTexSubImage2D()**

Color Table Event

```
// called when color table is changed in Papaya viewer
papaya.viewer.RapiscanVolume.prototype.changeColorTable = function (viewer, lutName) {
    this.colorTable = new papaya.viewer.ColorTable(lutName, true);
    // set intensity range
    this.updateLut(this.dataRangeMin, this.dataRangeMax, false);
    // set opacity range
    var max = Math.min(255, this.dataRangeMin + 2 * (this.dataRangeMax - this.dataRangeMin));
    this.updateLut(this.dataRangeMin, max, true);
    this.viewer.volumeLevelMin = this.dataRangeMin;
    this.viewer.volumeLevelMax = this.dataRangeMax;
    this.viewer.opacityMin = this.dataRangeMin;
    this.viewer.opacityMax = this.dataRangeMax;
};
```

Color Table Data Range

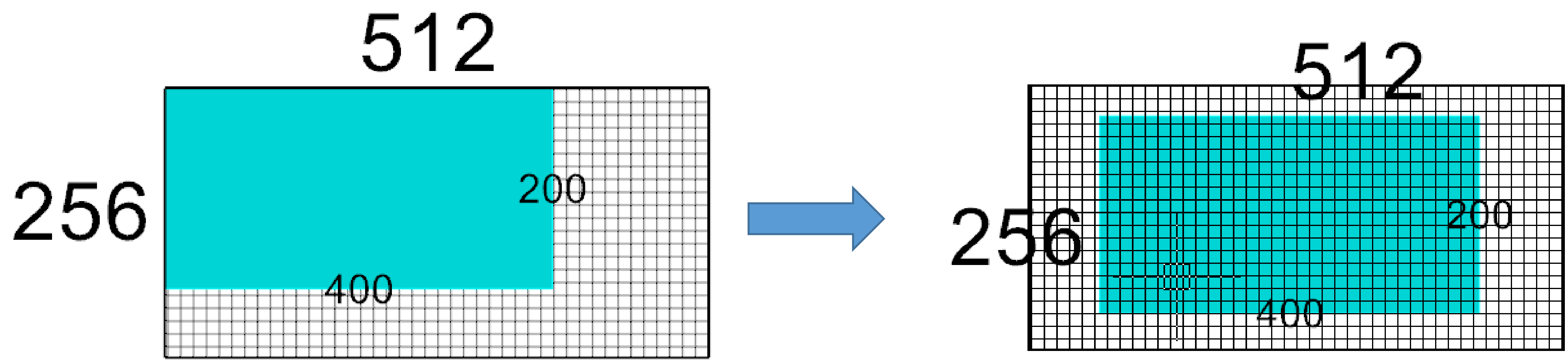
```
// determine significant range of the data
var max = 0;
var start = 32;
var maxIndex = start;
for (var i = start; i < 256; i++) {
    if (max < histogram[i]) {
        maxIndex = i;
        max = histogram[i];
    }
}
var l = maxIndex - 1;
while (l > 0 && histogram[l] > 0.01 * max) {
    l -= 1;
}
this.dataRangeMin = l;
l = maxIndex + 1;
while (l < 255 && histogram[l] > 0.01 * max) {
    l += 1;
}
this.dataRangeMax = l;
```

Data range is set from histogram in **BindVolume()**

Window/Level Event

```
papaya.viewer.Viewer.prototype.windowLevelChanged = function (windowChange, levelChange) {
    var range, step, minFinal, maxFinal;

    // compute change delta
    var windowWidth = this.volumeLevelMax - this.volumeLevelMin;
    step = Math.max(1.0, windowWidth * 0.025);

    // adjust window or level depending on horizontal/vertical mouse drag
    if (Math.abs(windowChange) > Math.abs(levelChange)) {
        minFinal = this.volumeLevelMin + (step * papaya.utilities.MathUtils.signum(windowChange));
        maxFinal = this.volumeLevelMax + (-1 * step * papaya.utilities.MathUtils.signum(windowChange));
    } else {
        minFinal = this.volumeLevelMin + (step * papaya.utilities.MathUtils.signum(levelChange));
        maxFinal = this.volumeLevelMax + (step * papaya.utilities.MathUtils.signum(levelChange));
    }

    if (maxFinal <= minFinal) { // do not allow window width to become zero
        minFinal = this.volumeLevelMin;
        maxFinal = this.volumeLevelMax;
    }
    else { // update current W/L setting
        this.volumeLevelMin = Math.max(0, Math.min(255, minFinal));
        this.volumeLevelMax = Math.max(this.opacityMin, Math.min(255, maxFinal));
    }

    // update volume view with current W/L
    this.volumeView.updateLut(this.volumeLevelMin, this.volumeLevelMax, false);
    this.drawViewer(true);
};
```

Handling NPOT Textures



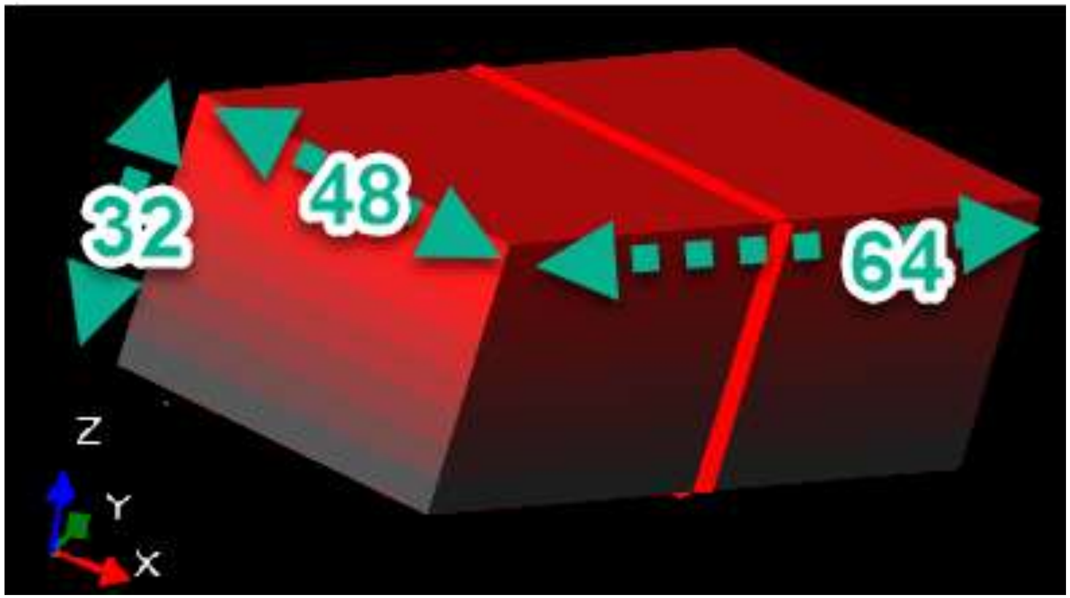
- Handle Non-Power-Of-Two texture dimensions
- Pad dimensions of allocated texture up to nearest POT
- Offset and Scale the texture coordinates to account for the “gap”
- Texture extents are 200/256 and 400/512 in X and Y i.e. (0.78, 0.78) instead of (1.0,1.0)
- Use offset of (1.0 – 0.78)/2 to centre the texture
- *Update: Not needed as NPOT driver support has improved*

updateVolumeView()

KB-UIS
We Work In-Depth

```
// pad texture dimensions up to nearest power-of-two
this.texSizeX = this.texSizeY = this.texSizeZ = 1;
while (this.texSizeX < this.volume.header.imageDimensions.cols) this.texSizeX *= 2;
while (this.texSizeY < this.volume.header.imageDimensions.rows) this.texSizeY *= 2;
while (this.texSizeZ < this.volume.header.imageDimensions.slices) this.texSizeZ *= 2;

// use offsets to centre the volume within the POT texture bounds
this.texScaleX = this.volume.header.imageDimensions.cols / this.texSizeX;
this.texScaleY = this.volume.header.imageDimensions.rows / this.texSizeY;
this.texScaleZ = this.volume.header.imageDimensions.slices / this.texSizeZ;
this.anisotropyX = 1.0;
this.anisotropyY = this.texSizeY/this.texSizeX;
this.anisotropyZ = this.texSizeZ/this.texSizeX;
var texExtentScale = 1.732;
var texOffset = (texExtentScale - 1.0) / 2.0;
this.volumeSliceTexCoords = new Float32Array([
    0.0, this.texScaleY * texExtentScale, 0.5 * this.texScaleZ * texExtentScale,
    0.0, 0.0, 0.5 * this.texScaleZ * texExtentScale,
    this.texScaleX * texExtentScale, this.texScaleY * texExtentScale, 0.5 * this.texScaleZ * texExtentScale,
    this.texScaleX * texExtentScale, 0.0, 0.5 * this.texScaleZ * texExtentScale
]);
for(var i =0;i<12;++i){
    this.volumeSliceTexCoords[i] -= texOffset;
}
```

Handling Anisotropy



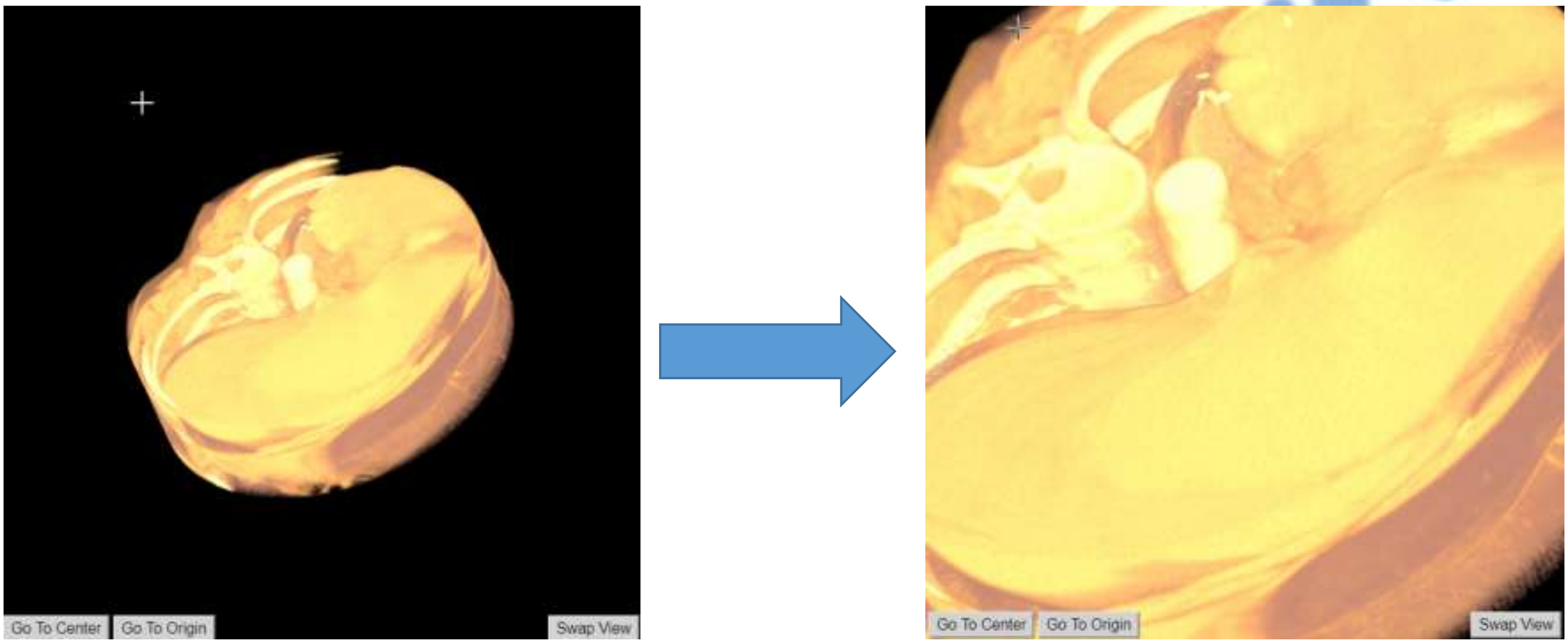
- Handle different spatial dimensions along volume dimensions i.e. Width≠Height≠Depth
- Texture coordinates always vary [0..1] across volume extents
- Solution: apply 3D scaling in transformation matrix

Texture Matrix

```
this.anisotropyX = 1.0;
this.anisotropyY = this.texSizeY/this.texSizeX;
this.anisotropyZ = this.texSizeZ/this.texSizeX;


mat4.identity(this.texMatrix);
mat4.scale(this.texMatrix, [1.0 / this.anisotropyX, 1.0 / this.anisotropyY, 1.0 / this.anisotropyZ]);
mat4.translate(this.texMatrix, [this.anisotropyX * this.texScaleX / 2.0,
    this.anisotropyY * this.texScaleY / 2.0,
    this.anisotropyZ * this.texScaleZ / 2.0]);
mat4.multiply(this.mouseRotDrag, this.mouseRotCurrent, this.mouseRotTemp);
this.mouseRotTemp[12] = this.mouseRotTemp[13] = this.mouseRotTemp[14] = 0.0;
mat4.inverse(this.mouseRotTemp);
mat4.multiply(this.texMatrix, this.mouseRotTemp, this.texMatrix);
mat4.translate(this.texMatrix, [-this.anisotropyX * this.texScaleX / 2.0,
    -this.anisotropyY * this.texScaleY / 2.0,
    -this.anisotropyZ * this.texScaleZ / 2.0]);
```

- Compute relative scale in each dimension (anisotropy factors)
- Scale the texture matrix by relative scale in each dimension (x,y,z)

Handling Zoom



- Alt + vertical mouse drag in volume view
- Handled in **Viewer.mouseDown** and **Viewer.mouseMove** – mouse movement delta is used to update **volumeView.zoom**
- **RapiscanVolume.updateView()** applies the zoom factor to the projection matrix in call to **mat4.ortho()**

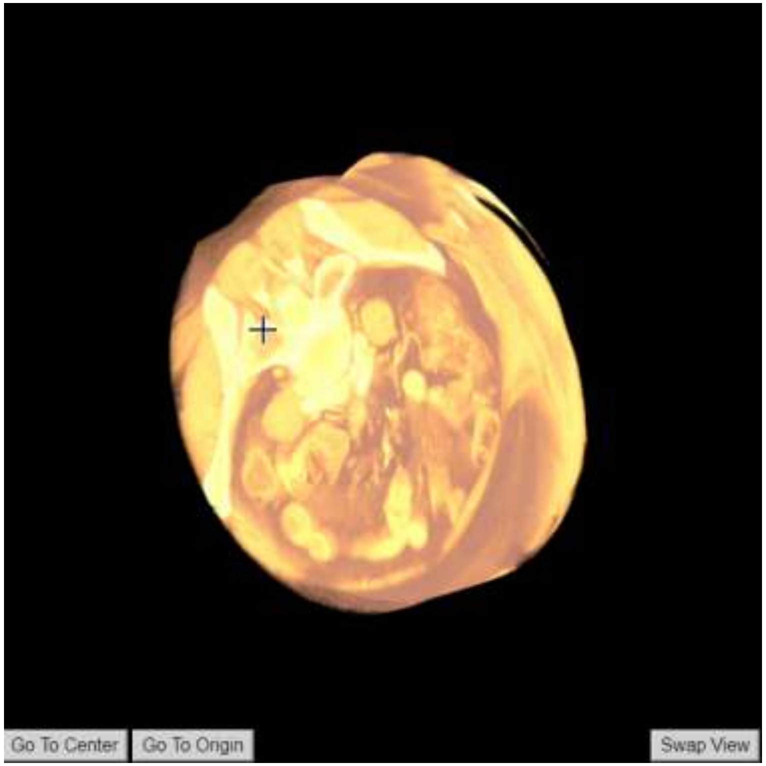
Viewer.mouseMove()

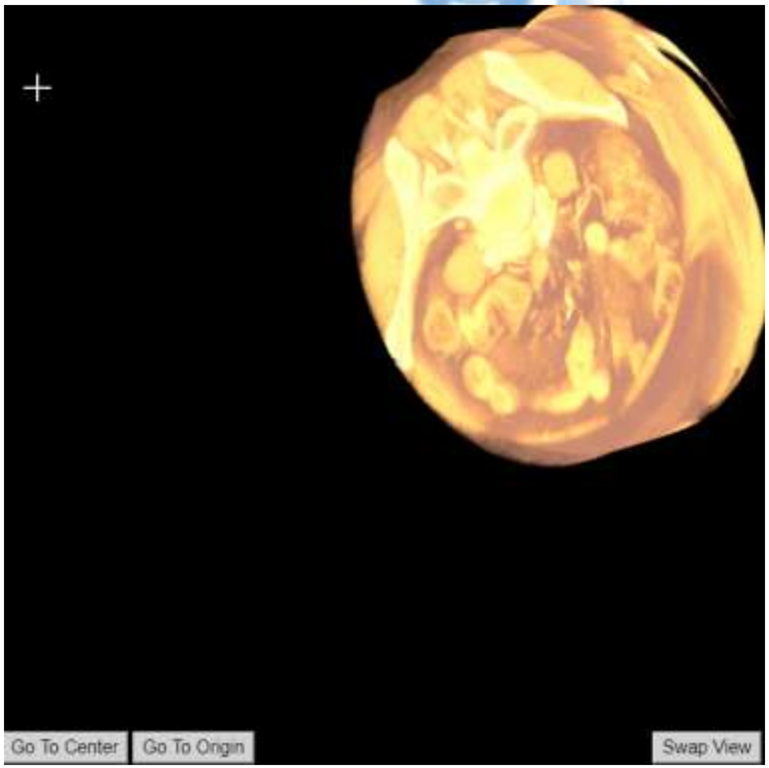
```
} else if (this.isZoomMode) {
    if (this.selectedSlice === this.volumeView) {
        this.volumeView.zoom += (currentMouseY - this.previousMousePosition.y) * 0.001; this.drawViewer(false, true);
        this.drawViewer(false, true);
        this.previousMousePosition.x = currentMouseX;
        this.previousMousePosition.y = currentMouseY;
    } else {
        zoomFactorCurrent = ((this.previousMousePosition.y - currentMouseY) * 0.05);
        this.setZoomFactor(this.zoomFactorPrevious - zoomFactorCurrent);

        this.axialSlice.updateZoomTransform(this.zoomFactor, this.zoomLocX, this.zoomLocY, this.panAmountX,
            this.panAmountY, this);
        this.coronalSlice.updateZoomTransform(this.zoomFactor, this.zoomLocX, this.zoomLocZ, this.panAmountX,
            this.panAmountZ, this);
        this.sagittalSlice.updateZoomTransform(this.zoomFactor, this.zoomLocY, this.zoomLocZ, this.panAmountY,
            this.panAmountZ, this);
    }

    this.drawViewer(true);
} else {
```

RapiscanVolume.updateView()

```
// set up orthographic projection
papaya.viewer.RapiscanVolume.prototype.updateView = function () {
    var size = this.yHalf * this.zoom;
    var offsetX = this.panX * this.zoom;
    var offsetY = this.panY * this.zoom;
    //this.pMatrix = mat4.ortho(-this.yHalf, this.yHalf, -this.yHalf, this.yHalf, -this.yHalf, this.yHalf);
    this.pMatrix = mat4.ortho(-size + offsetX, size + offsetX, -size + offsetY, size + offsetY, -size, size);
};
```

Handling Pan





- Alt + Shift + mouse drag in volume view
- Handled in **Viewer.mouseDown** and **Viewer.mouseMove** – mouse movement delta is used to update **volumeView.panX/Y**
- **RapiscanVolume.updateView()** applies the pan offset to the projection matrix in call to **mat4.ortho()**

Viewer.mouseMove()

```
else if (this.isPanning) {
  if (this.selectedSlice === this.volumeView) {
      var scale = this.volumeView.xDim / this.volumeView.screenDim;
      this.volumeView.panX += (this.previousMousePosition.x - currentMouseX)*scale;
      this.volumeView.panY += (currentMouseY - this.previousMousePosition.y) * scale;
      this.previousMousePosition.x = currentMouseX;
      this.previousMousePosition.y = currentMouseY;
      this.drawViewer(false, true);
  } else {
      this.setCurrentPanLocation(
          this.convertScreenToImageCoordinateX(currentMouseX, this.selectedSlice),
          this.convertScreenToImageCoordinateY(currentMouseY, this.selectedSlice),
          this.selectedSlice.sliceDirection
      );
  }
```

RapiscanVolume.updateView()

```
// set up orthographic projection
papaya.viewer.RapiscanVolume.prototype.updateView = function () {
    var size = this.yHalf * this.zoom;
    var offsetX = this.panX * this.zoom;
    var offsetY = this.panY * this.zoom;
    //this.pMatrix = mat4.ortho(-this.yHalf, this.yHalf, -this.yHalf, this.yHalf, -this.yHalf, this.yHalf);
    this.pMatrix = mat4.ortho(-size + offsetX, size + offsetX, -size + offsetY, size + offsetY, -size, size);
};
```

2D MPR Slices

- Handled in **papaya.viewer.ScreenSlice**
- **ScreenSlice.updateSlice()** computes the MPR slices
- Determines current “slice”, based on view and position
- Depending on MPR view, slice plane can be:
  - Axial – XY plane
  - Coronal – XZ plane
  - Sagittal – YZ plane
- Loop over XY / XZ / YZ volume indices
- Sample volume data using **Volume.getVoxelAtCoordinate**(i,j,k)

RGB MPR View

```
for (ctrY = 0; ctrY < this.yDim; ctrY += 1) {
    for (ctrX = 0; ctrX < this.xDim; ctrX += 1) {
        value = 0;
        thresholdAlpha = 255;
        layerAlpha = this.screenVolumes[ctr].alpha;

        if (rgb) {
            if (this.sliceDirection === papaya.viewer.ScreenSlice.DIRECTION_AXIAL) {
                value = this.screenVolumes[ctr].volume.getVoxelAtIndex(ctrX, ctrY, slice, timepoint, true);
            } else if (this.sliceDirection === papaya.viewer.ScreenSlice.DIRECTION_CORONAL) {
                value = this.screenVolumes[ctr].volume.getVoxelAtIndex(ctrX, slice, ctrY, timepoint, true);
            } else if (this.sliceDirection === papaya.viewer.ScreenSlice.DIRECTION_SAGITTAL) {
                value = this.screenVolumes[ctr].volume.getVoxelAtIndex(slice, ctrX, ctrY, timepoint, true);
            }

            index = ((ctrY * this.xDim) + ctrX) * 4;
            this.imageData[ctr][index] = value;

            this.imageDataDraw.data[index] = (value >> 16) & 0xff;
            this.imageDataDraw.data[index + 1] = (value >> 8) & 0xff;
            this.imageDataDraw.data[index + 2] = (value) & 0xff;
            this.imageDataDraw.data[index + 3] = thresholdAlpha;
```

No lookup – unpack RGB from 24-bit integer

Grayscale MPR View

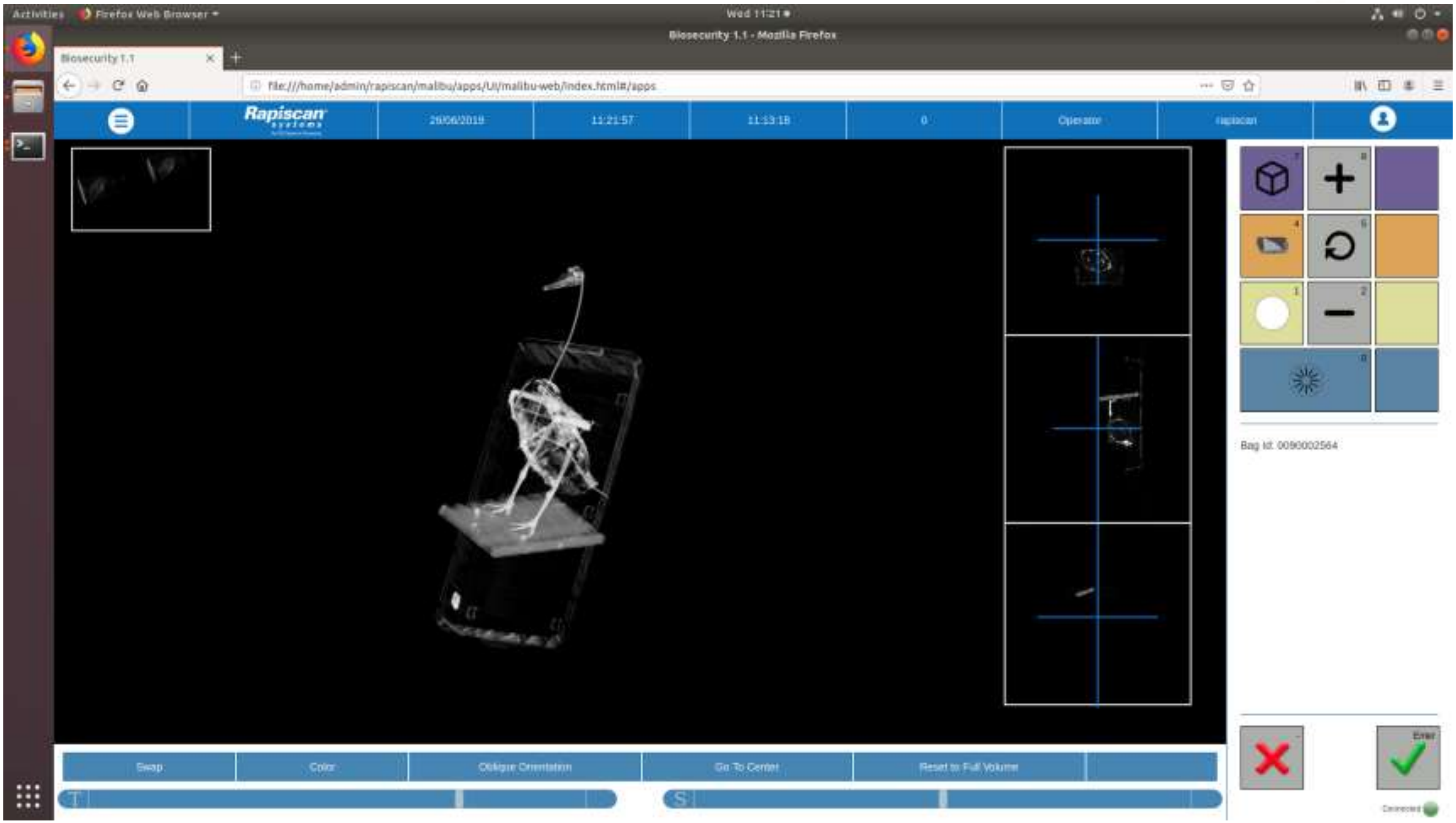
```
if (this.sliceDirection === papaya.viewer.ScreenSlice.DIRECTION_AXIAL) {
    value = this.screenVolumes[ctr].volume.getVoxelAtCoordinate((ctrX - origin.x) *
        voxelDims.xSize, (origin.y - ctrY) * voxelDims.ySize, (origin.z - slice) *
        voxelDims.zSize, timepoint, !interpolation);
} else if (this.sliceDirection === papaya.viewer.ScreenSlice.DIRECTION_CORONAL) {
    value = this.screenVolumes[ctr].volume.getVoxelAtCoordinate((ctrX - origin.x) *
        voxelDims.xSize, (origin.y - slice) * voxelDims.ySize, (origin.z - ctrY) *
        voxelDims.zSize, timepoint, !interpolation);
} else if (this.sliceDirection === papaya.viewer.ScreenSlice.DIRECTION_SAGITTAL) {
    value = this.screenVolumes[ctr].volume.getVoxelAtCoordinate((slice - origin.x) *
        voxelDims.xSize, (origin.y - ctrX) * voxelDims.ySize, (origin.z - ctrY) *
        voxelDims.zSize, timepoint, !interpolation);
}
```

Look up RGBA value from color table

```
this.imageDataDraw.data[index] = this.screenVolumes[ctr].colorTable.lookupRed(value, originalVal) * layerAlpha;
this.imageDataDraw.data[index + 1] = this.screenVolumes[ctr].colorTable.lookupGreen(value, originalVal) * layerAlpha;
this.imageDataDraw.data[index + 2] = this.screenVolumes[ctr].colorTable.lookupBlue(value, originalVal) * layerAlpha;
this.imageDataDraw.data[index + 3] = thresholdAlpha;
```

DAY 3

BIO 3D Viewer



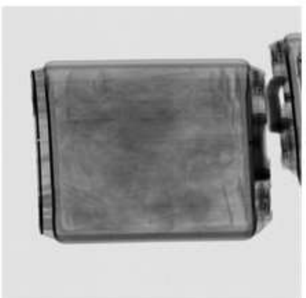
- Websocket-based stream instead of DICOS
- See Git repository for current code snapshot

BIO 3D Viewer

- Javascript/WebGL viewer receives streaming input not from DICOS files
- 3D volume chunks stream on websocket
- 2D dual-view scrolling image chunks stream on websocket
- By default dual-view scrolling image view is active
- Volume is loaded once all chunks are received
- Inset preview thumbnail is displayed once volume is loaded

3D Viewer

2D Scrolling Image Data

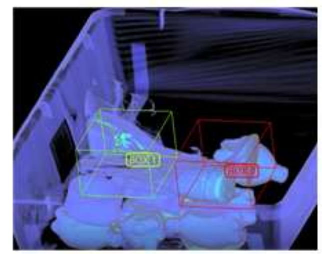


- IMAGE_SCROLL→
Receive incoming pixel data
Update 2D Image Buffer
Refresh scrolling view

Volume Slice Data

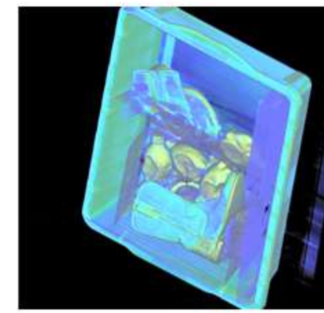
- VOLUME_LOAD_CHUNK→
Load incoming slice data
- VOLUME_LOAD_FINISH→
Reconstruct volume
Update GL Texture Buffer

Box and Voxel Markup



- BOX_INSERT, BOX_DELETE, BOX_RESIZE, BOX_POSITION, BOX_TEXT→Update overlays
- VOXEL_MARKUP→Update volume markup mask

User Input/Events



- VIEW_LAYOUT→Toggle and configure views: 3D Volume, 2D Scrolling, 2D MPR, VOI Inset
- VIEW_SET, ROTATE,PAN,ZOOM→Update transforms
- VOLUME_RENDERED, SHADED_SURFACE→Update render mode
- WINDOW/LEVEL, COLOR_TABLE→Update LUT
- VOI_ENABLE,VOI_INSET→ Update clip region

Key Methods

RapiscanVolume

- onLoadChunk()
- onDualViewUpdate()
- onVolumeChunkUpdateMultiThreaded()
- loadPapayaVolume()
- onUserEvent()
- onUserQuery()
- drawScene()
- renderVolume() / renderImage()
- drawBoxOverlay()

Dual-View Image Scrolling

- Javascript/WebGL viewer receives streaming input instead of DICOS files
- 3D volume chunks stream on websocket
- 2D dual-view scrolling image chunks stream on websocket
- By default dual-view scrolling image view is active
- Volume is loaded once all chunks are received
- Inset preview thumbnail is displayed once volume is loaded

2D Scrolling Image Display

- **initImageShaders**(): load 2D shaders
- **onDualViewUpdate**(): handle incoming dual-view chunk - add it to the queue for processing
- **updateDualViewTexture**(): update dual-view textured image based on new incoming image data
- **bindDualViewImage**(): create dual view image 2D texture and rectangle
- **renderImage**(): draw 2D textured image
- **startScrollTimer**: scrolls the dual-view image by refreshing at regular intervals

2D Image Streaming

```
// timer is used to scroll the dual-view images
var timer = [false, false];
var timerVar = [null, null];
var startScrollTimer = function (that, view) {
    timerVar[view] = setInterval(function () {
        if (that.volumeViewType !== papaya.viewer.Viewer.VOLUME_VIEW_3D) {
            that.viewer.drawViewer();
        }
    }, that.scrollInterval);
}

// process queued dual-view chunks to update scrolling image at regular intervals
papaya.viewer.RapiscanVolume.prototype.processQueuedChunks = function (view) {
    if (!timer[view] && queuedDualViewChunks[view].length > 0) {
        timer[view] = true;
        var that = this;
        setTimeout(function () {
            startScrollTimer(that, view);
        }, 100);
    }
}
```

```
// handle incoming dual-view chunk - add it to the queue for processing
onDualViewUpdate = function (view, chunkIndex, cols, chunkRows, pixels) {
  queuedDualViewChunks[view].push({ data: pixels, rows: chunkRows });
  if (chunkIndex == 0) {
    this.dualViewRows[view] = this.dualViewColumns[view] = cols;
    this.currentChunkSlice = [0, 0];
    this.dualViewImageTexture = [null, null];

    if (!this.initialized) {
      this.isVolumeReceived = false;
      this.volume = new papaya.volume.Volume(this.viewer.container.display, this.viewer,
                    this.viewer.container.params);
      if (this.volumeTexture !== null) this.releaseVolume(this.context);
      this.voxelBytes = 2;
      this.xDim = this.volume.xDim = 16;
      this.yDim = this.volume.yDim = 16;
      this.zDim = this.volume.zDim = 16;
      this.chunkedVolumeSliceCount = 0;
      this.viewer.volume = this.volume;
      this.viewer.initializeViewer();
    }
  }
  if (chunkIndex > 2) {
    this.processQueuedChunks(view);
  }
}
```

Vertex Shader

```
uniform mat4 uPMatrix;
uniform float uScrollOffset;

in vec3 aVertexPosition;
in vec2 aVertexTexCoord;

out vec2 vVertexTexCoord;

void main(void) {
  vVertexTexCoord = aVertexTexCoord + vec2(0.0, uScrollOffset);
  gl_Position = uPMatrix * vec4(aVertexPosition, 1.0);
}
```

Fragment Shader

```
uniform sampler2D uImageSampler; // dual-view image texture
uniform sampler2D uLutSampler; // lookup table texture (256x1 2D texture)
uniform bool uBackgroundBlend;
in vec2 vVertexTexCoord;
out vec4 fragColor;

float intensity(vec4 rgba)
{
   return rgba.g + rgba.b/16.0 + rgba.a/256.0;
}

void main(void) {
     fragColor = texture(uImageSampler, vVertexTexCoord);
     float lutIndex = intensity(fragColor);
     fragColor = texture(uLutSampler, vec2(lutIndex, 0.0));
     if(!uBackgroundBlend) fragColor.a = 1.0;
}
```

renderImage()

```
this.updateView(false); // set view for 2D image

gl.disable(gl.DEPTH_TEST);
// blending needed for background blend (if any)
gl.enable(gl.BLEND);
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);

// clear entire view with border color
gl.enable(gl.SCISSOR_TEST);
if (this.volumeViewType !== 0) {
    switch (view) {
        case 0: gl.scissor(0, gl.viewportHeight - viewportHeight, viewportWidth,
                        viewportHeight); break;
        case 1: gl.scissor(viewportWidth, gl.viewportHeight - viewportHeight, viewportWidth,
                        viewportHeight); break;
    }
    gl.clearColor(this.mainBorderColor[0], this.mainBorderColor[1], this.mainBorderColor[2],
        this.mainBorderColor[3]);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
}
```

renderImage()

```
// restrict draw area to within border
var border = 2;
switch (view) {
    case 0: gl.scissor(border, gl.viewportHeight - viewportHeight + border, viewportWidth -
        2 * border, viewportHeight - 2 * border); break;
    case 1: gl.scissor(viewportWidth, gl.viewportHeight - viewportHeight + border,
        viewportWidth - border, viewportHeight - 2 * border); break;
}

// clear area within border using background color
gl.clearColor(this.backgroundColor[0], this.backgroundColor[1], this.backgroundColor[2],
        this.backgroundColor[3]);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
switch (view) {
    case 0: gl.viewport(border, gl.viewportHeight - viewportHeight + border,
        viewportWidth - border, viewportHeight - border); break;
    case 1: gl.viewport(viewportWidth + border, gl.viewportHeight - viewportHeight +
        border, viewportWidth - border, viewportHeight - border); break;
}
```

renderImage()

```
if (this.dualViewImageTexture[view] !== null) {
    // set up shader and texture
    gl.disable(gl.CULL_FACE);
    gl.useProgram(this.imageShader);
    gl.activeTexture(view == 0 ? gl.TEXTURE2 : gl.TEXTURE3);
    gl.bindTexture(gl.TEXTURE_2D, this.dualViewImageTexture[view]);
    gl.uniform1i(this.imageShader.imageSamplerUniform, view == 0 ? 2 : 3);
    gl.activeTexture(gl.TEXTURE0);
    gl.bindTexture(gl.TEXTURE_2D, this.lutTexture);
    gl.uniform1i(this.imageShader.lutSamplerUniform, 0);

    // apply current scroll offset (effectively shifting/scrolling the image)
    gl.uniform1f(this.imageShader.scrollOffsetUniform, chunkScrollOffset[view]);
    gl.uniform1i(this.imageShader.backgroundBlend, this.backgroundColor[0] > 0 ||
        this.backgroundColor[1] > 0 || this.backgroundColor[2] > 0);

    // set up project matrix so that image fills the view
    var halfsizeX = this.dualViewColumns[view] / 2.0;
    var halfsizeY = this.dualViewRows[view] / 2.0;
    var pMat = mat4.ortho(-halfsizeX, halfsizeX, -halfsizeY, halfsizeY, -halfsizeY, halfsizeY);
    gl.uniformMatrix4fv(this.imageShader.pMatrixUniform, false, pMat);
```

renderImage()

```
    // bind 2D texture corresponding to the dual-view image and draw

    gl.bindBuffer(gl.ARRAY_BUFFER, this.imageVertAttributesBuffer[view]);
    gl.enableVertexAttribArray(this.imageShader.vertexPositionAttribute);
    gl.vertexAttribPointer(this.imageShader.vertexPositionAttribute, 3, gl.FLOAT, false, 20, 0);
    gl.enableVertexAttribArray(this.imageShader.vertexTexCoordAttribute);
    gl.vertexAttribPointer(this.imageShader.vertexTexCoordAttribute, 2, gl.FLOAT, false, 20, 12);

    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);

    gl.disableVertexAttribArray(this.imageShader.vertexPositionAttribute);
    gl.disableVertexAttribArray(this.imageShader.vertexTexCoordAttribute);
  }

  gl.disable(gl.SCISSOR_TEST);

  // flag current view as being up-to-date
  dualViewTextureUpdated[view] = false;
}
```

onVolumeChunkUpdateMultiThreaded()

```
if (chunkIndex == 0 && !isLoadingVolume) isReceivingChunks = true;

// do not interfere with 2D streaming to avoid lags
if (isLoadingVolume || !isReceivingChunks) return;

// starting new volume
if (chunkIndex == 0) {
   this.chunkedVolumeSliceCount = 0;
   this.viewer.clearBoxes();
}

// add volume chunks to queue
queuedVolumeChunks.push({ data: pixels, slices: chunkSlices });
transferables.push(pixels.buffer);
chunkSliceCounts.push(chunkSlices);
this.chunkedVolumeSliceCount += chunkSlices;
```

onVolumeChunkUpdateMultiThreaded()

```
// final chunk received – load the volume
if (endOfBag) {
    this.isVolumeReceived = false;
    this.volume = new papaya.volume.Volume(this.viewer.container.display,
                    this.viewer, this.viewer.container.params);
    if (this.volumeTexture !== null) { // unload previous volume
       this.releaseVolume(this.context);
    }
    this.voxelBytes = 2;
    this.xDim = this.volume.xDim = cols;
    this.yDim = this.volume.yDim = rows;
    this.zDim = this.volume.zDim = this.chunkedVolumeSliceCount;
    histogram = new Uint32Array(65536);
    this.viewer.volume = this.volume;
    this.viewer.initializeViewer();
```

onVolumeChunkUpdateMultiThreaded()

```
// more volume setup
 if (this.volumeData === null || currentAlloc < this.xDim * this.yDim * this.zDim) {
        currentAlloc = Math.max(512, this.xDim) * Math.max(512, this.yDim) * Math.max(1024,
            this.zDim);
        this.volumeData = new Uint16Array(currentAlloc);
        console.log("onVolumeChunkUpdateMultiThreaded: current allocation = ", currentAlloc);
     }
     this.volume.imageData.data = this.volumeData;

     // shared data for web worker
     transferables.push(this.volumeData.buffer);
     transferables.push(histogram.buffer);

     isLoadingVolume = true;
     isReceivingChunks = false;
     this.initVolume();
     this.texScaleX = this.xDim / this.texSizeX;
     this.texScaleY = this.yDim / this.texSizeY;
     this.texScaleZ = this.zDim / this.texSizeZ;
     this.xHalf = (this.xDim * this.xSize) / 2.0;
     this.yHalf = (this.yDim * this.ySize) / 2.0;
     this.zHalf = (this.zDim * this.zSize) / 2.0;
```

onVolumeChunkUpdateMultiThreaded()

```
// setup volume-load web worker
var that = this;

function createWorker() {
   var v = new Worker('worker/volumeWorker.js');
   v.postMessage({ chunks: transferables, chunkSliceCounts: chunkSliceCounts, sliceSize:
       that.xDim * that.yDim, volumeAlloc: currentAlloc }, transferables);

   queuedVolumeChunks = [];
   transferables = [];
   chunkSliceCounts = [];

   v.onmessage = function (e) {
      console.log('loaded volume');
      that.volumeData = that.volume.imageData.data = e.data.volumeArray;
      histogram = e.data.histogram;
      computedMaximum = e.data.maximum;
      that.loadPapayaVolumeMultiThreaded(computedMaximum);
      isLoadingVolume = false;
   };
}
// spawn the web worker to load the volume without blocking the 2D stream
createWorker();
```

Volume-load Web Worker

```
self.onmessage = function (e) {
   var chunkArrayLength = e.data.chunks.length;
   var outVolumeArray = new Uint16Array(e.data.chunks[chunkArrayLength - 2]);
   var outHistogram = new Uint32Array(e.data.chunks[chunkArrayLength - 1]);
   var maximum = 0;
   var volumeIndex = 0;
   // process volume to compute histogram (can be avoided if provided in data stream)
   for (var c = 0; c < e.data.chunkSliceCounts.length; ++c) {
      var chunkData = new Uint16Array(e.data.chunks[c]);
      var pixels = chunkData.subarray(5, chunkData.length);
      var pixelCount = e.data.chunkSliceCounts[c] * e.data.sliceSize;
      for (var pixelIndex = 0; pixelIndex < pixelCount; ++pixelIndex) {
         var pixVal = pixels[pixelIndex];
         outVolumeArray[volumeIndex++] = pixVal;
         ++outHistogram[pixVal];
         maximum = Math.max(maximum, pixVal);
      }
   }
   // post the output data
   self.postMessage({ maximum: maximum, histogram: outHistogram, volumeArray: outVolumeArray
            }, [outHistogram.buffer, outVolumeArray.buffer]);
   close();
}
```

onUserEvent

```
// handle user events - used to interface with the parent application
papaya.viewer.RapiscanVolume.prototype.onUserEvent = function (eventMessage) {
    switch (eventMessage.eventID) {
        case "SET_VIEW_TYPE":
            this.setViewType(eventMessage.eventData);
            break;
        case "SET_VOLUME_RENDER_MODE":
            this.setRenderMode(eventMessage.eventData);
            break;
        case "SET_VIEW_ORIENTATION":
            this.setViewOrientation(eventMessage.eventData);
            break;
        case "SET_COLOR_TABLE_BY_NAME":
            this.changeColorTable(this.viewer, eventMessage.eventData);
            this.viewer.drawViewer(true);
            break;
        case "SET_COLOR_TABLE_BY_LUT":
            this.setLUT(this.viewer, eventMessage.eventData);
            this.viewer.drawViewer(true);
```

- Set eventID and eventData accordingly to communicate UI events and initiate desired actions by the 2D/3D Viewer
- See rapiscan-viewer/rapiscanvolume-interface.xlsx for documentation of events

onUserQuery

```
// handle user queries and return requested state info
papaya.viewer.RapiscanVolume.prototype.onUserQuery = function (eventMessage) {
    switch (eventMessage.eventID) {
        case "GET_BOXES":
            return this.getBoxes();
        case "GET_BOX_COUNT":
            return this.viewer.boxCount;
        case "GET_ACTIVE_BOX":
            return this.viewer.currentBoxIndex;
    }
}
```

- Set eventID accordingly to receive desired state information/data from the 2D/3D Viewer

Ongoing and Future Work

- Separate 2D (Dual-View/Scrolling) Image functionality from Volume viewer
  - Currently 2D and 3D views share the WebGL context, LUT
  - Need for displaying 2D views across multiple canvases
- Support additional Data Formats (Z-Effective etc.)
- Additional LUT/Color-Table formats
- 3D Box-Picking

Questions

Send questions to:

3d@kbvis.com

Resources

- Computer Graphics: Principles and Practice, Foley, van Dam, Hughes
- CS 410: Introduction to Computer Graphics, Colorado State University
- WebGL: Up and Running, Tony Parisi, O'Reilly Press
- WebGL Beginner's Guide, Diego Cantor and Brandon Jones
- Papaya DICOM Viewer
- Papaya source code on GitHub
- http://dicomiseasy.blogspot.com

Thank You !