
Chapter 1:03
Basic Rendering
Principia v3.0
Reference Series
(c) Western Star
Entertainment Ltd, 2003-2006

Chapter 3 – Basic Rendering

- Rendering is the process whereof images are formed from graphic primitive data.
- Principia provides many components to hold graphic primitive data (such as geometry or materials) and to manage its rendering (such as objects and world)
- This chapter introduces the basic rendering components and explains the basic principles behind their use.

Chapter 3 – Basic Rendering

- **Pre-requisite knowledge:**
  - Conceptual understanding of 3D rendering
  - Conceptual understanding of cameras and lights
  - Conceptual understanding of graphic primitive data
  - Conceptual understanding of GPUs and their states
  - No need for graphics programming background
- **If you find Chapter 3 rather steep, or discover that small but important details such as GPU states or vertex formats are taken for granted:**
  - Chapter 11 provides a broader discussion of basics in its presentation of the fixed function pipeline
  - Read the notes from an intro course on 3D graphics
  - Read the Microsoft DirectX vs7/vs8/vs9 tutorials

Chapter 3 – Basic Rendering

- **Central concepts in rendering**
  - Rendering framework
  - Graphic primitive data
  - Rendering sequence
- **Principia utilizes the system GPU(s) to render**
  - It relies on a software layer such as DirectX™ or OpenGL™ to issue commands to the GPU.
  - It encapsulates graphic primitive data into components configured in script, thereby saving you the hardship of managing graphics data.
  - It encapsulates rendering sequences in the script flow. This enables the creation of rich and complete visual artistry, without exposing the user to the intricacies of graphics programming.

Chapter 3 – Frameworks

- **Rendering framework:**
  - A structured way to organize and present different graphic data to construct the final image of an application object.
- **Principia features several rendering frameworks:**
  - The *2A* framework is based obsolete 2D blit renders. It is not deemed worthy of coverage here.
  - The *3A* framework is the general-purpose Principia V3 framework, designed for mass commercial production.
  - The *4A* framework adds high-order primitive modeling and many powerful specialized rendering effects.
- **Each framework has its own components, and strikes a distinctive tradeoff between quality, performance, flexibility and complexity.**

Chapter 3 – Frameworks

The Principia *3A* rendering framework

- Each scene is processed on the CPU (server) and converted into a sequence (queue) of graphic data and operations to be executed by the GPU (client). This is referred to as “API processing”.
- Once the sequence is closed, the GPU locks it and processes the data and operations thereon, this forming an image. This is “GPU processing”.
- The API and GPU steps can be run synchronously (at every frame, the CPU generates the queue, and then the GPU executes it) or asynchronously.
- Geometry is represented as buffers of 3D polygon primitives, rasterized by the client GPU(s).
- GPU processing of graphic data is controlled by (a) shaders and/or (b) fixed function unit states.

Chapter 3 – Frameworks

- **The Principia *3A* rendering framework**
  - We recommend using programmable (shader-based) rendering for production.
  - Chapter 3 uses solely the *3A* framework, with a mix of programmable and fixed rendering. The legacy fixed pipeline is covered in Chapter 11.
- **Pros and Cons of *3A***
  - Versatility
  - Separate Api/Gpu sequences optimize performance and leverage modern multi-processor hardware.
  - Limitation to triangle-based primitives
  - No support for cutting-edge rendering methods such as ray-tracing or simulations on the GPU.

Chapter 3 – Primitives

- **Graphic primitive major types:**
  - Executable: components that are rendered to form the image (e.g. vertex streams or blitted surfaces).
  - State: components that provide context and configuration to the GPU as it renders executable primitives (e.g. materials and their numerous sub-components, kinexes, cameras, lights... etc.).
  - Operational: components that force the GPU or CPU to perform an operation on other primitives (e.g. effects, layer operations... etc.).
- **Principia provides an unparalleled array of graphic primitives, encapsulated across several families of components.**

Chapter 3 – Primitives

- AX_Grafic components in the *3A* framework
  - GX_Color – contains color data
  - GX_Font3A – contains font glyphs and data
  - GX_Light3A – contains lighting data
  - GX_Camera3A – contains view settings
  - GX_Surface – contains image data
  - GX_VShader3A – contains vertex shaders
  - GX_PShader3A – contains pixel shaders
  - GX_FShader3A – contains effect shaders
  - GX_Material3A – contains material data
  - GX_VSet3A – contains vertex streams (geometry)
  - GX_Mesh3A – contains renderable assemblies

Chapter 3 – Primitives

- Other *3A* rendering components in Chapter 3
  - Kinexes
  - Effects
- Kinexes
  - Hold data on how primitives should be positioned
  - Hold data on which keyframes to render
  - Exposed at the AX_Object level
- Effects
  - Operate on AX_Grafic* components
  - Executed on the GPU as part of scheduled material
- There are many more components involved in rendering that will be covered at later time.

Chapter 3 – Sequences

- **Rendering sequences describe the order in which primitives get scheduled for execution.**
- **The *3A* framework does not force rendering sequences, except for few simple rules:**
  - Scenes (Chap.2) and few other specialized controls generate frame loops and GPU instruction queues via the Principia interfaces.
  - AX_Control components present and encapsulate renderable items (and/or other AX_Controls) within a scene.
  - AX_World components encapsulate 3D renderable objects in streams ordered in sequential layers.
  - Materials, effects... can be inserted at any point of the rendering sequence because many components have material sockets.
  - The rendering sequence is thus driven by the organization of controls in the script and by the population of world layers.
  - Advanced developers can over-ride and/or complement the rendering sequence from within the user code via API calls.

Chapter 3 – Sequences

- **Rendering sequences are paramount**
  - Determine what you will see
  - Determine performance
  - Determine experience
- **Highest level concepts recap**
  - Rendering framework
  - Graphic primitives (rendering components)
  - Rendering sequence
- **Chapter 3 introduces the very basics of the rendering process using Principia components. It also describes some key elements that you need to master to achieve good results.**

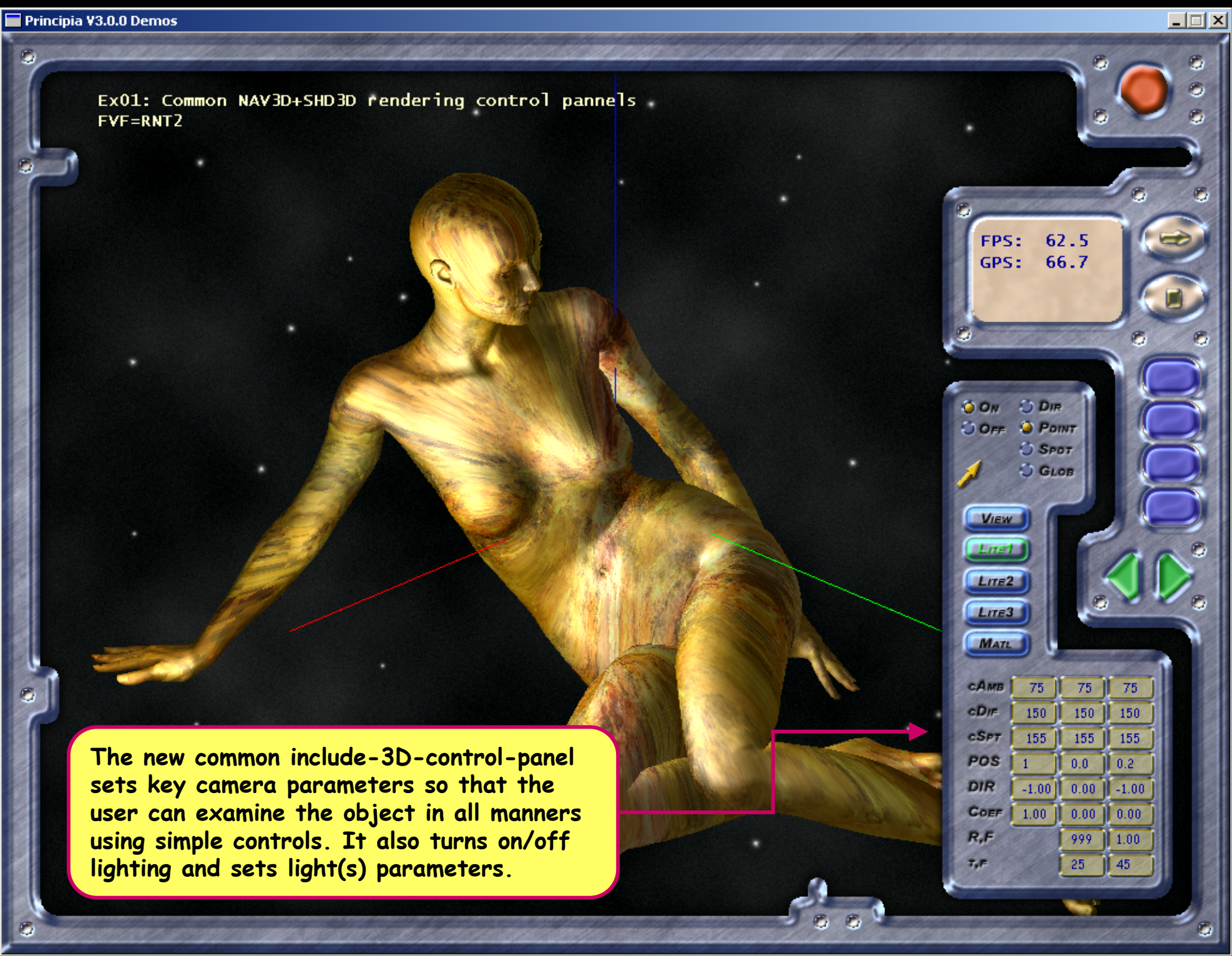
Chapter 3 – Roadmap

- Rendering is a vast and complex topic!
- Chapter 3 begins with the basics of how to render objects that are relatively plain and simple. The focus is on the basic “how?”.
- Chapter 4 describes how to design shaders and manage data to achieve the results you want.
- Chapter 5 describes the basic methods for making your images come to life.
- Chapters 7 and 8 describe the high-level data structures that make graphics come alive as worlds, characters, objects and more.
- And that is only the beginning…

Chapter 3 – Roadmap

- Demo_103A - Unified Rendering GUI
- Demo_103B - Geometry
- Demo_103C - Materials
- Demo_103D - Cameras and Lights
- Demo_103E - Using Shaders
- Demo_103F - Frame Buffers
- Demo_103G - Depth-Stencils Buffers
- Demo_103H - Render Targets
- Demo_103I - Textures
- Demo_103J - Texture Coordinates
- Demo_103K - Anti-Aliasing
- Demo_103L - GPU Fixed Stages Use


Chapter 1:03
Basic Rendering
Principia v3.0
Reference Series
(c) Western Star
Entertainment Ltd, 2003-2006


Principia V3.0.0 Demos
Ex01: Common NAV3D+SHD3D rendering control pannels
FVF=RNT2
FPS: 62.5
GPS: 66.7
On
Off
Dir
Point
Spot
Glob
View
Lite1
Lite2
Lite3
Matl
cAmb 75 75 75
cDif 150 150 150
cSpt 155 155 155
POS 1 0.0 0.2
DIR -1.00 0.00 -1.00
Coef 1.00 0.00 0.00
R,F 999 1.00
T,F 25 45
The new common include-3D-control-panel sets key camera parameters so that the user can examine the object in all manners using simple controls. It also turns on/off lighting and sets light(s) parameters.

D103A – Unified GUI

- **Before jumping into rendering itself, let's extend the common GUI from Chapter 1**
  - Enable interactive control of camera
  - Enable interactive setting of up to three lights and of their illumination mode
  - Enable this control panel to work with both shaders and the fixed function pipeline
  - Encapsulate into includable script(s)
- **This would provide a good example of more complex integration between control and rendering components.**

D103A – Unified GUI

- **File encapsulation**
  - Module_NAV_3D.cfg/cpp: Data variables and control components for the 3D control panel itself
  - Module_SHD_3D.cfg/cpp: Data variables and materials for connecting 3DCP panel data to shader registers
  - Script defines 3D control panel components
  - User code manages the data and control states
- **Albeit the complexity of these modules is much bigger than anything done so far, 3DCP is created using the concepts from Chapters 1/2.**
- **Unfamiliar with the lighting and camera data structures? Read D103D and Chapter 11.**

D103A – Unified GUI

- **Our demo need only to include the 3DCP modules, define materials and geometry, and render !**

```
##parse "../Demo_Common/Files_Scripts/Module_GUIDEM.cfg"
##parse "../Demo_Common/Files_Scripts/Module_NAV_3D.cfg"
##parse "../Demo_Common/Files_Scripts/Module_SHD_3D.cfg"

##define (S_EX01) as <SURFACE>
   #tag Definer   =  (       DEF_TEX )
   #tag File      =  "../Demo_Common/Files_Media/Textures/Tex_Metal_BurnishedCopper_512.bmp"

##define (M_EX01) as <MATERIAL_3A>
   #tag RS      =  (               RS_FILLMODE ,        RA_SOLID )
   . . .
   #tag RS      =  (       RS_ALPHABLENDENABLE ,              0 )
   #tag TS      =  (                TS_COLOROP ,     TO_MODULATE ,   0 )
   #tag TS      =  (              TS_COLORARG1 ,      TA_TEXTURE ,   0 )
   #tag TS      =  (              TS_COLORARG2 ,      TA_DIFFUSE ,   0 )
   #tag TS      =  (                TS_ALPHAOP ,     TO_MODULATE ,   0 )
   #tag TS      =  (              TS_ALPHAARG1 ,      TA_TEXTURE ,   0 )
   #tag TS      =  (              TS_ALPHAARG2 ,      TA_DIFFUSE ,   0 )
   #tag VShader =  (                     NONE )
   #tag PShader =  (                     NONE )
   #tag TX      =  (               S_EX01 , 0 )

##define (G_EX01) as <VERTEXSET_3A>
   #tag File      =  "../Demo_Common/Files_Media/Meshes/Mesh_Sculpture_FemaleA.3ds"
   #tag Definer   =  ( VTD_FVF_FILE )
   #tag Indexed   =  (          YES )

##define (R_EX01) as <MESH_3A>
   #tag Component =  (           0 ,    M_EX01 ,  G_EX01 )

##define (O_EX01) as <GENOBJECT_3A>
   #tag Construct =  (  "Decl(M);RotZ(300.0);Mul(1.5,1.5,1.5);Mov(0.0,0.2,-0.5);" ,    R_EX01 ,       NONE )

##define (WRLD_EX01) as <WORLD_3A>
   #tag Layer_Decl  =  (     L_ITEMS ,   "ITEMS" , - , - , 16 )
   #tag Layer_Item  =  (     L_ITEMS ,                O_EX01 )
   #tag Layer_Item  =  (     L_ITEMS ,             O_AXISYS )

##define (VIEW_EX01) as <WORLDVIEWER_3A>
   #tag World     =  (     WRLD_EX01 )
   #tag Camera    =  (   CAMERA_ST3D )
```

D103A – Unified GUI

- **Likewise, the user code needs only to include the 3DCP modules and make sure they are called in the scene init / main user procedures.**

```
#define  CONF_FILE    "Demo_103A_Main.cfg"
#define  ROOT_NAME    "..//Product_040701_Demos//Demo_1.03.A_UnifiedGUI//"
#include              "../Demo_Common/Files_Code/Module_GUIDEM.cpp"
#include              "../Demo_Common/Files_Code/Module_NAV_3D.cpp"
#include              "../Demo_Common/Files_Code/Module_SHD_3D.cpp"


void User_Scene_Init (void* VArg)
BGN_ACTOR_CODE{

   /* Execute initialization for the common display components */
   Common_PerformanceDisplay_Init(VArg);
   Common_ControlPannel3D_Init(VArg);
   Common_ShaderConnect3D_Init(VArg);

END_ACTOR_CODE}


void User_Scene_Main (void* VArg)
BGN_ACTOR_CODE{

   /* Execute scene loop operations for the common display components */
   Common_PerformanceDisplay_Main(VArg);
   Common_ControlPannel3D_Main(VArg);
   Common_ShaderConnect3D_Main(VArg);

END_ACTOR_CODE}
```

- **There is no need for code to render! But what is behind the scenes? It is worth your while to take a look how data and controls are integrated…**

D103A – Unified GUI

- **NAV3D script architecture. It pays off to organize content when building complex functional units:**
  - Variables holding lighting parameters and choices
  - Predefined lights (the parameters are over-ridden in user code by connecting to variables above)
  - Predefined default material (parameters over-ridden in user code by connecting to variables above)
  - Standard 2D/3D cameras and their keyboard controllers
  - Fonts, panel surfaces and component frames
  - Pages for the View/Light1/Light2/Light3/Material tabs. Preceded by its ingredients as follows (a) buttons (b) sliders (c) input boxes (d) checklists.
  - Control panel book. The book definition is preceded by that of its switching buttons and a sub-page to keep them together.
  - Standard axis object
  - Standard GPU reset renderable object
- **Only select highlights from the script will be listed next. The full script is just multiple instances of these themes.**

D103A – Unified GUI

- **NAV3D implementation: Joint slider-input box. Note box focus and VARMOD methods.**

```
##define (BT_NVW1) as <BUTTON_2C>
   #tag State     =  (    NORMAL ,              F_NAV3D_S ,     NONE ,        NONE ,         - )
   #method        =  (   ON_OVER ,      CH_CRS ,   X_OVER ,   NORMAL ,   F_NAV3D_S ,      NONE ,        - ,        - )
   #method        =  (   ON_LHLD ,      CH_NAV ,   X_NMV1 ,   NORMAL ,   F_NAV3D_S ,      NONE ,        - ,        - )
   #method        =  (   ON_LHLD ,      CH_NAV ,   X_IVM1 ,   NORMAL ,   F_NAV3D_S ,      NONE ,        - ,        - )
   #tag Region . . .

##define (SLD_NVW1) as <SLIDERBOX_2A>
   #tag Anchor    =  (     - ,     - )
   #tag Region    =  (    32 ,   282 ,   141 ,   292 )
   #tag VarMin    =  (       VA_DIST )
   #tag VarVal    =  (       VV_DIST )
   #tag VarMax    =  (       VB_DIST )
   #tag Delta     =  (           0.1 )
   #tag CoDecBtn  =  (          NONE ,     - ,     - )
   #tag CoMovBtn  =  (       BT_NVW1 ,   -11 ,   -10 )
   #tag CoIncBtn  =  (          NONE ,     - ,     - )
   #tag CoIndctr  =  (          NONE ,     - ,     - )
   #method        =  (       ON_OVER ,      CH_CRS ,     X_OVER  )
   #method        =  (       ON_LCLK ,      CH_NAV ,     X_NSV1  )
   #method        =  (       ON_LCLK ,      CH_NAV ,     X_IVM1  )
   #method        =  (      EXE_SETV ,      CH_NAV ,     X_NSV1  )
   #method        =  (      EXE_MOVE ,      CH_NAV ,     X_NMV1  )

##define (INB_NVW1) as <INPUTBOX_2A>
   #tag Variable  =  (       VV_DIST )
   #tag TextBox   =  (   150 ,   280 ,   185 ,   295 )
   #tag TextIns   =  (   155 ,   287 )
   #tag MaxLen    =  (            32 )
   #tag PlateOn   =  (          NULL ,     - ,     - )
   #tag PlateOff  =  (          NULL ,     - ,     - )
   #tag FontOn    =  (   FN_SMLTXT_R )
   #tag FontOff   =  (   FN_SMLTXT_B )
   #tag VarFmt    =  "%.1f"
   #tag OverRead  =  (            NO )
   #tag Jumpout   =  (           YES )
   #tag Dynamic   =  (            NO )
   #method        =  (       ON_OVER ,      CH_CRS ,   X_HIDE    )
   #method        =  (       ON_LCLK ,      CH_NAV ,   X_IVU1    )
   #method        =  (    EXE_UPDATE ,      CH_NAV ,   X_IVU1    )
   #method        =  (       ON_LCLK ,      CH_NAV ,   X_IVF1    )
   #method        =  (     EXE_FOCUS ,      CH_NAV ,   X_IVF1    )
   #method        =  (    EXE_VARMOD ,      CH_NAV ,   X_IVM1    )
```

D103A – Unified GUI

- **NAV3D implementation: Clickable selectors with pre-built images. States are monitored in user code.**

```
##define (BTN_LITE) as <BUTTON_2C>
   #tag State     =  (    CHEKED ,   F_NAV3D_E ,     NONE ,     NONE ,           - )
   #tag State     =  (    UNCHEK ,   F_NAV3D_N ,     NONE ,     NONE ,           - )
   #method        =  (   ON_OVER ,      CH_CRS ,   X_OVER ,   UNCHEK ,    F_NAV3D_N ,       NONE ,        - )
   #method        =  (   ON_LCLK ,      CH_NAV ,   X_CHEK ,   UNCHEK ,    F_NAV3D_N ,  A_BEEP_01 ,        - )
   #method        =  (   ON_OVER ,      CH_CRS ,   X_OVER ,   CHEKED ,    F_NAV3D_E ,       NONE ,        - )
   #method        =  (   ON_LCLK ,      CH_NAV ,   X_UNCH ,   CHEKED ,    F_NAV3D_E ,  A_BEEP_01 ,        - )
   #method        =  ( EXE_STATE ,      CH_NAV ,   X_CHEK ,   CHEKED )
   #method        =  ( EXE_STATE ,      CH_NAV ,   X_UNCH ,   UNCHEK )

##define (BTN_SPEC) as <BUTTON_2C>
   #tag State     =  (    UNCHEK ,   F_NAV3D_N ,     NONE ,     NONE ,           - )
   #tag State     =  (    CHEKED ,   F_NAV3D_E ,     NONE ,     NONE ,           - )
   #method        =  (   ON_OVER ,      CH_CRS ,   X_OVER ,   UNCHEK ,    F_NAV3D_N ,       NONE ,        - )
   #method        =  (   ON_LCLK ,      CH_NAV ,   X_SHEK ,   UNCHEK ,    F_NAV3D_N ,  A_BEEP_01 ,        - )
   #method        =  (   ON_OVER ,      CH_CRS ,   X_OVER ,   CHEKED ,    F_NAV3D_E ,       NONE ,        - )
   #method        =  (   ON_LCLK ,      CH_NAV ,   X_SNCH ,   CHEKED ,    F_NAV3D_E ,  A_BEEP_01 ,        - )
   #method        =  ( EXE_STATE ,      CH_NAV ,   X_SHEK ,   CHEKED )
   #method        =  ( EXE_STATE ,      CH_NAV ,   X_SNCH ,   UNCHEK )

##define (CKL_NVW1) as <CHECKLIST_2A>
   #tag MaxSelect =  (           1 )
   #tag LayoutX   =  (     0 ,     0 )
   #tag LayoutY   =  (     0 ,    20 )
   #tag Plate     =  (          NONE ,     - ,     - )
   #method        =  (       ON_OVER ,      CH_CRS ,    X_OVER   )
   #method        =  (       ON_LCLK ,      CH_CRS ,    X_OVER   )
   #tag Item      =  (  SEL1 ,     0 )
   #tag Media     =  (     F_NAV3D_N ,     0 ,     0 ,         F_NAV3D_Y ,        NONE,    NONE )
   #tag Text      =  (            "" ,     0 ,     0 ,                "" ,        NONE,    NONE )
   #tag Item      =  (  SEL2 ,     1 )
   #tag Media     =  (     F_NAV3D_N ,     0 ,     0 ,         F_NAV3D_Y ,        NONE,    NONE )
   #tag Text      =  (            "" ,     0 ,     0 ,                "" ,        NONE,    NONE )
```

D103A – Unified GUI

- **NAV3D implementation: The multiple standalone checkboxes require simpler pointer focus methods. Thus, the images are pre-built on the page and blank regions are used to drive interaction.**

```
##define (INB_L1AG) as <INPUTBOX_2A>
   #tag Variable  =  (       VL_L1AG )
   #tag TextBox   =  (     0 ,     0 ,    35 ,    18 )
   #tag TextIns   =  (     5 ,     9 )
   #tag MaxLen    =  (           32 )
   #tag PlateOn   =  (         NULL ,     - ,     - )
   #tag PlateOff  =  (         NULL ,     - ,     - )
   #tag FontOn    =  (   FN_SMLTXT_R )
   #tag FontOff   =  (   FN_SMLTXT_B )
   #tag VarFmt    =  "%4d"
   #tag OverRead  =  (          YES )
   #tag Jumpout   =  (           NO )
   #tag Dynamic   =  (           NO )
   #method        =  (       ON_OVER ,      CH_CRS ,   X_HIDE    )
   #method        =  (       ON_LCLK ,      CH_NAV ,   X_IL1V    )
   #method        =  (    EXE_UPDATE ,      CH_NAV ,   X_IL1V    )

. . .


##define (PG_NAV3D_L1) as <PAGE_2A>
   #tag Anchor    =  (             - ,      - )
   #tag Region    =  (             - ,      - ,       - ,      - )
   #tag Element   =  (    F_NAV3D_F1 ,      - ,       - ,      - )
   #tag Element   =  (      CKL_NL1T ,      - ,       - ,      - )
   #tag Element   =  (      CKL_NL1S ,      - ,       - ,      - )
   #tag Element   =  (      INB_L1AR ,     78 ,     263 ,      - )
   #tag Element   =  (      INB_L1AG ,    118 ,     263 ,      - )
   #tag Element   =  (      INB_L1AB ,    156 ,     263 ,      - )
```

D103A – Unified GUI

- **NAV3D implementation: Light source components. The defined parameters are overwritten in the user code.**

```
##define (LITE_STD1) as <LIGHT_3A>
   #tag Type      =  (       LIGHT_DIRECT )
   #tag ColAmbi   =  (      1,    1,    1 )
   #tag ColDiff   =  (      1,    1,    1 )
   #tag ColSpec   =  (      0,    0,    0 )
   #tag Position  =  (    0.0,  0.0,  0.0 )
   #tag Direction =  (    0.0,  0.0,  0.0 )
   #tag Range     =  (               999.9 )
   #tag Falloff   =  (                 1.0 )
   #tag Theta     =  (                25.0 )
   #tag Phi       =  (                45.0 )
   #tag Coeffs    =  ( 1.0, 0.0, 0.0, 0.0 )

. . .

##define (LITE_STD3) as <LIGHT_3A>
   #tag Type      =  (       LIGHT_SPOT   )
   #tag ColAmbi   =  (      1,    1,    1 )
   #tag ColDiff   =  (      1,    1,    1 )
   #tag ColSpec   =  (      0,    0,    0 )
   #tag Position  =  (    0.0,  0.0,  0.0 )
   #tag Direction =  (    0.0,  0.0,  0.0 )
   #tag Range     =  (               999.9 )
   #tag Falloff   =  (                 1.0 )
   #tag Theta     =  (                25.0 )
   #tag Phi       =  (                45.0 )
   #tag Coeffs    =  ( 1.0, 0.0, 0.0, 0.0 )
```

- **Note that lights, camera and material color components are only used when rendering on the fixed function pipeline. When using shaders, key properties of these components are copied to register-mapped variables (see Chapter 4).**

D103A – Unified GUI

- **NAV3D implementation: Default material, resetting the GPU to a desired neutral state. Some of these parameters are reset in the user code based on the control panel state.**

```
##define (M_STDNULL) as <MATERIAL_3A>
   #tag RS       = (               RS_ZENABLE ,              1 )
   #tag RS       = (           RS_ZWRITEENABLE ,              1 )
   #tag RS       = (         RS_STENCILENABLE ,              0 )
   #tag RS       = (             RS_DEPTHBIAS ,         0.0000 )
   #tag RS       = (              RS_FILLMODE ,       RA_SOLID )
   #tag RS       = (              RS_CULLMODE ,      RA_CULLCCW )
   #tag RS       = (     RS_POINTSPRITEENABLE ,              0 )
   #tag RS       = (      RS_POINTSCALEENABLE ,              0 )
   #tag RS       = (           RS_VERTEXBLEND ,      BL_DISABLE )
   #tag RS       = (       RS_NORMALIZENORMALS ,              1 ) ‘ DO NOT FORGET THIS ONE, OR ALL YOUR FFPL RENDERS WILL BOMB!!!
   #tag RS       = (            RS_LOCALVIEWER ,              1 )
   #tag TS       = (                TS_COLOROP ,     TO_MODULATE ,   0 )
   #tag TS       = (              TS_COLORARG1 ,      TA_TEXTURE ,   0 )
   #tag TS       = (              TS_COLORARG2 ,      TA_DIFFUSE ,   0 )
  . . .
   #tag TS       = (              TS_ALPHAARG2 ,      TA_CURRENT ,   1 )
   #tag RS       = (  RS_AMBIENTMATERIALSOURCE ,     RA_MATERIAL )
   #tag RS       = (  RS_DIFFUSEMATERIALSOURCE ,       RA_COLOR1 )
   #tag RS       = ( RS_SPECULARMATERIALSOURCE ,       RA_COLOR2 )
   #tag RS       = ( RS_EMISSIVEMATERIALSOURCE ,     RA_MATERIAL )
   #tag FVFAmb   = (    255,  255,  255 ,  255 )
   . . .
   #tag FVFPwr   = (                     1.00 )
   #tag RS       = (        RS_ALPHATESTENABLE ,              0 )
   #tag RS       = (               RS_ALPHAREF ,              0 )
   #tag RS       = (              RS_ALPHAFUNC ,      CA_GREATER )
   #tag RS       = (       RS_ALPHABLENDENABLE ,              1 )
   #tag RS       = (              RS_SRCBLEND ,     FA_SRCALPHA )
   #tag RS       = (             RS_DESTBLEND ,  FA_INVSRCALPHA )
   #tag SA       = (              SS_ADDRESSU ,         SA_WRAP ,   0 )
   #tag SA       = (              SS_ADDRESSV ,         SA_WRAP ,   0 )
   #tag SA       = (             SS_MAGFILTER ,         SA_NONE ,   0 )
   . . .
   #tag TS       = (          TS_TEXCOORDINDEX ,              0 ,   0 )
   . . .
   #tag TX       = (                NONE , 0 )
   . . .
   #tag PShader  = (                   NONE )
   #tag VShader  = (                   NONE )
   #tag FShader  = (                   NONE )
   #tag Phase    = (                     0 )
```

D103A – Unified GUI

- **NAV3D implementation: Standard GX_Camera3A linked to control variables (set via the control panel) and a signaling keyboard controller that activates camera motion methods.**

```
##define (KJAY_CAMS3D)as <KJAY_2A>
   #method        =  (    ON_KEYPRS  ,    CH_NAV ,     X_PANU  ,     AUP )
   #method        =  (    ON_KEYPRS  ,    CH_NAV ,     X_PAND  ,     ADN )
   #method        =  (    ON_KEYPRS  ,    CH_NAV ,     X_PANL  ,     ALF )
   #method        =  (    ON_KEYPRS  ,    CH_NAV ,     X_PANR  ,     ARG )

##define (CAMERA_ST3D)as <CAMERA_3A>
   #tag Target             =  (              NONE ) ' Render on the back buffer
   #tag Viewport           =  (  - ,  - ,   - ,   - ) ' Use full extent of render target
   #tag ViewVar.Perspec    =  (           VV_PERS )
   #tag ViewVar.Dist       =  (           VV_DIST )
   #tag ViewVar.Elev       =  (           VV_ELEV )
   #tag ViewVar.Azim       =  (           VV_AZIM )
   #tag ViewVar.Angle      =  (           VV_ANGL )
   #tag ViewVar.LookPos    =  (  VV_LATX ,  VV_LATY ,  VV_LATZ )
   #tag View.UpVec         =  (      0.0 ,      0.0 ,      1.0 )
   #tag View.Aspect        =  (              1.0 )
   #tag View.ZNear         =  (              0.0 )
   #tag View.ZFar          =  (            255.0 )
   #tag XfProj             =  (             NONE )
   . . .
   #tag MaterialBgn        =  (         M_STDNULL )
   #tag MaterialEnd        =  (                 - )
   #tag VsReg_View         =  (              49 )
   #tag Light              =  (         LITE_STD1 ,        1 )
   #tag Light              =  (         LITE_STD2 ,        0 )
   #tag Light              =  (         LITE_STD3 ,        0 )

   #tag UserCtl_PanW       =  (              200 )
   #tag UserCtl_PanH       =  (              150 )
   . . .

   #method                 =  (    EXE_PANU ,    CH_NAV ,     X_PANU )
   #method                 =  (    EXE_PAND ,    CH_NAV ,     X_PAND )
   #method                 =  (    EXE_PANL ,    CH_NAV ,     X_PANL )
   #method                 =  (    EXE_PANR ,    CH_NAV ,     X_PANR )
   . . .
```

D103A – Unified GUI

- **A simple 2D direct camera is provided for rendering flat images using transformed pixel-sized geometries.**

```
. . .
   #method               =  (    EXE_PANU ,    CH_NAV ,     X_PANU )
   #method               =  (    EXE_PAND ,    CH_NAV ,     X_PAND )
   #method               =  (    EXE_PANL ,    CH_NAV ,     X_PANL )
   #method               =  (    EXE_PANR ,    CH_NAV ,     X_PANR )
   #method               =  (   EXE_DISTN ,    CH_NAV ,    X_DISTN )
   #method               =  (   EXE_DISTF ,    CH_NAV ,    X_DISTF )
   #method               =  (   EXE_ZOOMN ,    CH_NAV ,    X_ZOOMN )
   #method               =  (   EXE_ZOOMF ,    CH_NAV ,    X_ZOOMF )
   #method               =  (   EXE_ELEVU ,    CH_NAV ,    X_ELEVU )
   #method               =  (   EXE_ELEVD ,    CH_NAV ,    X_ELEVD )
   #method               =  (   EXE_AZIML ,    CH_NAV ,    X_AZIML )
   #method               =  (   EXE_AZIMR ,    CH_NAV ,    X_AZIMR )
   #method               =  (     ON_PANS ,    CH_NAV ,     X_ULKX )
   #method               =  (     ON_PANS ,    CH_NAV ,     X_ULKY )
   #method               =  (     ON_PANS ,    CH_NAV ,     X_ULKZ )
   #method               =  (     ON_DIST ,    CH_NAV ,     X_IVM1 )
   #method               =  (     ON_ZOOM ,    CH_NAV ,     X_IVM4 )
   #method               =  (     ON_ELEV ,    CH_NAV ,     X_IVM2 )
   #method               =  (     ON_AZIM ,    CH_NAV ,     X_IVM3 )

##define (CAMERA_ST2D) as <CAMERA_3A>
   #tag Viewport         =  (  - ,   - ,   - ,  - )
   #tag ViewSim2D        =  (                1 )
   #tag MaterialBgn      =  (          M_STDNULL )
   #tag MaterialEnd      =  (                - )
   #tag VsReg_View       =  (               49 )
```

- **Some objects tags can take variables instead of values.**
  - **If the variable changes, the object is automatically updated without need of any user code. This is how we move the camera here...**

D103A – Unified GUI

- **NAV3D implementation: Much artwork is produced by third party art pipeline tools, and loaded from a standard 3D format such as 3DS (more on this in the next demo).**
  - **Two Principia 3DS loaders are provided with typical flag settings: the first welds all fragments in the 3DS file into single vertex streams without embedded materials, the second leaves the fragment material and SMG detail at the cost of having a structure-rich vertex stream.**

```
##define (STDMAXLDR_FUSED) as <PROCDAT_IMPORT3DS>
   #tag Transform         = "Decl(M); Mul(1.00, 1.00, 1.00);"
   #tag Ldo_MeshFacetize  = (   NO )
   #tag Ldo_MeshUseSmg    = (   NO )
   #tag Ldo_MeshUseMtl    = (   NO )
   #tag Ldo_FragNormals   = (   NO ,         NO )
   #tag Ldo_FragMeshWeld  = (       WELD_POSUV , 0.0 )
   #tag Asm_XSmg          = (  YES , WELD_POSUV , 0.0 )
   #tag Asm_XMat          = (  YES , WELD_POSUV , 0.0 )
   #tag Asm_XObj          = (   NO , WELD_NONE  , 0.0 )
   #tag Asm_Normals       = (   NO ,        YES )
   #tag Vgn_StripMtl      = (  YES )

##define (STDMAXLDR_FRAGS) as <PROCDAT_IMPORT3DS>
   #tag Transform         = "Decl(M); Mul(1.00, 1.00, 1.00);"
   #tag Ldo_MeshFacetize  = (   NO )
   #tag Ldo_MeshUseSmg    = (  YES )
   #tag Ldo_MeshUseMtl    = (  YES )
   #tag Ldo_FragNormals   = (   NO ,        YES )
   #tag Ldo_FragMeshWeld  = (       WELD_POSUV , 0.0 )
   #tag Asm_XSmg          = (  YES , WELD_POSUV , 0.0 )
   #tag Asm_XMat          = (   NO , WELD_POSUV , 0.0 )
   #tag Asm_XObj          = (   NO , WELD_NONE  , 0.0 )
   #tag Asm_Normals       = (   NO ,         NO )
   #tag Vgn_StripMtl      = (   NO )
```

D103A – Unified GUI

- **NAV3D user code: This is how we connect data references to components defined in script. Knowing the API data structure, we can access the data via these references …**

```
/* PageAA: obtain pointers to the Principia objects defined in the script */
aMAT     = (GX_Material3A*)   Principia->GetReferenceTo("M_STDNULL");
aCAM     = (GX_Camera3A*)     Principia->GetReferenceTo("CAMERA_ST3D");

/* PageVV: obtain pointers to the Principia objects defined in the script */
aBtnLITE = (CX_Button2C*)     Principia->GetReferenceTo("BTN_LITE");
aBtnSPEC = (CX_Button2C*)     Principia->GetReferenceTo("BTN_SPEC");
aCklPERS = (CX_CheckList2A*)  Principia->GetReferenceTo("CKL_NVW1");
aVarPERS = (SX_Variable*)     Principia->GetReferenceTo("VV_PERS");

/* PageL1: obtain pointers to the Principia objects defined in the script */
aL1      = (GX_Light3A*)      Principia->GetReferenceTo("LITE_STD1");
aCklNL1S = (CX_CheckList2A*)  Principia->GetReferenceTo("CKL_NL1S");
aCklNL1T = (CX_CheckList2A*)  Principia->GetReferenceTo("CKL_NL1T");
aVarL1AR = (SX_Variable*)     Principia->GetReferenceTo("VL_L1AR");
aVarL1AG = (SX_Variable*)     Principia->GetReferenceTo("VL_L1AG");
```

D103A – Unified GUI

- **NAV3D user code main: This is how we test single button states, checklist states and connect them to variable changes or render state modifications.**

```
/* View control: light enabler */
SelN = Principia->GetNamedValue("CHEKED");
Test = aBtnLITE->TestState(SelN, SelIDX);
if (Test) then{
   aMAT->SetRS(D3DRS_LIGHTING  ,          1 );
}else{
   aMAT->SetRS(D3DRS_LIGHTING  ,          0 );
endif}
Test = aBtnSPEC->TestState(SelN, SelIDX);
if (Test) then{
   aMAT->SetRS(D3DRS_SPECULARENABLE ,     1 );
}else{
   aMAT->SetRS(D3DRS_SPECULARENABLE ,     0 );
endif}

/* View control: perspective selection */
Test = aCklPERS->TestSelection(Sel1, SelIDX);
if (Test) then{
   aVarPERS->Assign(1);
   aMAT->SetRS(D3DRS_LOCALVIEWER ,        1 );
}else{
   aVarPERS->Assign(0);
   aMAT->SetRS(D3DRS_LOCALVIEWER ,        1 );
endif}
```

D103A – Unified GUI

- **NAV3D user code main: This is how we set a list of light object parameters. The light object will update itself when the camera that references it will be applied.**

```
/* View control: Light1 parameters */
aVarL1AR->Get(aL1->ColorAmbi->cR);
aVarL1AG->Get(aL1->ColorAmbi->cG);
aVarL1AB->Get(aL1->ColorAmbi->cB);
aVarL1DR->Get(aL1->ColorDiff->cR);
aVarL1DG->Get(aL1->ColorDiff->cG);
aVarL1DB->Get(aL1->ColorDiff->cB);
aVarL1SR->Get(aL1->ColorSpec->cR);
aVarL1SG->Get(aL1->ColorSpec->cG);
aVarL1SB->Get(aL1->ColorSpec->cB);
aVarL1PX->Get(aL1->PosX         );
aVarL1PY->Get(aL1->PosY         );
aVarL1PZ->Get(aL1->PosZ         );
aVarL1DX->Get(aL1->DirX         );
aVarL1DY->Get(aL1->DirY         );
aVarL1DZ->Get(aL1->DirZ         );
aVarL1C1->Get(aL1->AttC0        );
aVarL1C2->Get(aL1->AttC1        );
aVarL1C3->Get(aL1->AttC2        );
aVarL1RR->Get(aL1->Range        );
aVarL1FF->Get(aL1->Falloff      );
aVarL1TH->Get(aL1->Theta        );
aVarL1FI->Get(aL1->Phi          );
```

D103A – Unified GUI

- **NAV3D User code main. This is how we set the type of light and determine which camera lights are on or off.**

```
/* View control: Light3 modality */
Test = aCklNL3T->TestSelection(Sel1, SelIDX);
if (Test) aL3->Type = LIGHTTYPE_DIRECT;
Test = aCklNL3T->TestSelection(Sel2, SelIDX);
if (Test) aL3->Type = LIGHTTYPE_POINT;
Test = aCklNL3T->TestSelection(Sel3, SelIDX);
if (Test) aL3->Type = LIGHTTYPE_SPOT;
Test = aCklNL3T->TestSelection(Sel4, SelIDX);
if (Test) aL3->Type = LIGHTTYPE_GLOBAL;

/* View control: Light3 activity */
Test = aCklNL3S->TestSelection(Sel1, SelIDX);
if (Test) aCAM->LightsOn->ePtr[2] = 1;
Test = aCklNL3S->TestSelection(Sel2, SelIDX);
if (Test) aCAM->LightsOn->ePtr[2] = 0;
```

- **We are going to skip the rest of the light and material property updates, because they really apply for fixed function pipeline renders anyway. We just want to show the techniques for user code component management.**

D103A – Unified GUI

- **Useful note: When compiled in diagnostic mode, Principia generates a running log of issues and events in the core dump file as it runs scenes and parses scripts.**

```
Parsing file: [Demo_103A_Main.cfg]
Parsing file: [../Demo_Common/Files_Scripts/Module_GUIDEM.cfg]
Parsing file: [..\..\Principia_Library\Principia_StandardDefinitions_Core_v300.cfg]
Parsing file: [..\..\Principia_Library\Principia_StandardDefinitions_Utility_v300.cfg]
Parsing file: [..\..\Principia_Library\Principia_StandardDefinitions_Procedures_v300.cfg]
Parsing file: [../Demo_Common/Files_Scripts/Module_INTERFACES.cfg]
Unrecognized tag: DRMAgent
Unrecognized tag: RepeatRate
Unrecognized tag: HoldMoves
Tag similarity: Device_HW, Device_HWRASTERIZER
Tag similarity: Device_HW, Device_HWTNL
Tag similarity: Device_DP2, Device_DP2EX
Tag similarity: DevDcl_UBYTE4, DevDcl_UBYTE4N
Tag similarity: VtxProcCaps_TEXGEN, VtxProcCaps_TEXGENSPHERE
Tag similarity: Ztest_NE, Ztest_NEVER
Tag similarity: Stencil_INCR, Stencil_INCRSAT
Tag similarity: Stencil_DECR, Stencil_DECRSAT
. . .
Tag similarity: TexOp_MODULATE, TexOp_MODULATE2X
Tag similarity: TexOp_MODULATE, TexOp_MODULATE4X
Tag similarity: TexAdress_MIRROR, TexAdress_MIRRORONCE
Tag similarity: VolAdress_MIRROR, VolAdress_MIRRORONCE
Parsing file: [../Demo_Common/Files_Scripts/Module_NAV_3D.cfg]
MT: Set: Expand: (32)
Tag similarity: CoInd, CoIndctr
Unrecognized tag: CoIndctr
Parsing file: [../Demo_Common/Files_Scripts/Module_SHD_3D.cfg]

3DSFILE: Fna(../Demo_Common/Files_Media/Meshes/Mesh_Sculpture_FemaleA.3ds)
CH:0x4D4D: Ver(3)
CH:0x3D3E: Len(10)
CH:0x100: Len(10)

CH:0x4000: Obj(default)
CH:0x4100: Len(1363758)
MT: Set: Expand: (96021)
```

D103A – Unified GUI

- **SHD3D implementation. The script which enables to connect control panel data to shader registers begins by defining the transfer variables.**

```
##define <SYS_VARIABLE>  =  (  CSHDR_LPOS_A ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Principal light position
##define <SYS_VARIABLE>  =  (  CSHDR_LDIR_A ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Principal light direction
##define <SYS_VARIABLE>  =  (  CSHDR_AMBI_A ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Principal light ambient
##define <SYS_VARIABLE>  =  (  CSHDR_DIFF_A ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Principal light diffuse
##define <SYS_VARIABLE>  =  (  CSHDR_SPEC_A ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Principal light specular
##define <SYS_VARIABLE>  =  (  CSHDR_LPAR_A ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Principal light parameters
##define <SYS_VARIABLE>  =  (  CSHDR_LPOS_B ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Secondary light position
##define <SYS_VARIABLE>  =  (  CSHDR_LDIR_B ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Secondary light direction
##define <SYS_VARIABLE>  =  (  CSHDR_AMBI_B ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Secondary light ambient
##define <SYS_VARIABLE>  =  (  CSHDR_DIFF_B ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Secondary light diffuse
##define <SYS_VARIABLE>  =  (  CSHDR_SPEC_B ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Secondary light specular
##define <SYS_VARIABLE>  =  (  CSHDR_LPAR_B ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Secondary light parameters
##define <SYS_VARIABLE>  =  (  CSHDR_AMBI_M ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'General purpose, material ambient
##define <SYS_VARIABLE>  =  (  CSHDR_DIFF_M ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'General purpose, material diffuse
##define <SYS_VARIABLE>  =  (  CSHDR_SPEC_M ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'General purpose, material specular
##define <SYS_VARIABLE>  =  (  CSHDR_EMIT_M ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'General purpose, material emissive
##define <SYS_VARIABLE>  =  (  CSHDR_PARS_M ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'General purpose, material power
##define <SYS_VARIABLE>  =  (    CSHDR_EPOS ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Camera position
##define <SYS_VARIABLE>  =  (    CSHDR_EDIR ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Camera direction
##define <SYS_VARIABLE>  =  (    CSHDR_EPAR ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Camera parameters
##define <SYS_VARIABLE>  =  (    CSHDR_TIME ,   VARTYPE_FSR ,  0.00, 0.00, 0.00, 0.00 ) 'Camera parameters
```

D103A – Unified GUI

- **SHD3D script also provides two key materials:**
  - **StdCamlit: to be used in vertex shaders. Maps camera, light and other non-local data to various vertex shader registers.**
  - **StdPerPixel: to be used for per-pixel lighting. Maps primary light ARGB properties to pixel shader registers 1,2 and 3.**
  - **These included materials enable us to unburden the demo scripts and to use a consistent shader data structure.**

```
##define (MSHD_STDCAMLIT) as <MATERIAL_3A>
   #tag VsMat     =  (           4 ,      MAT_VPT ) 'World-to-projection space transform in shader register 4
   #tag VsMat     =  (          12 ,       MAT_VT ) 'World-to-camera space transform in shader register 12
   #tag VsMat     =  (          16 ,       MAT_PT ) 'Projection transform in shader register 16
   #tag VsVar     =  (          30 , CSHDR_LPOS_A ) 'Light A position
   #tag VsVar     =  (          31 , CSHDR_LDIR_A ) 'Light A direction
   #tag VsVar     =  (          32 , CSHDR_LPAR_A ) 'Light A parameter
   #tag VsVar     =  (          33 , CSHDR_AMBI_A ) 'Light A ambient color
   #tag VsVar     =  (          34 , CSHDR_DIFF_A ) 'Light A diffuse color
   #tag VsVar     =  (          35 , CSHDR_SPEC_A ) 'Light A specular color
   #tag VsVar     =  (          36 , CSHDR_LPOS_B ) 'Light B position
   . . .
   #tag VsVar     =  (          41 , CSHDR_SPEC_B ) 'Light B specular color
   #tag VsVar     =  (          42 , CSHDR_AMBI_M ) 'Substrate material ambient
   . . .
   #tag VsVar     =  (          46 , CSHDR_PARS_M ) 'Substrate material power
   #tag VsVar     =  (          47 , CSHDR_EPOS )   'Important geometry: camera position
   #tag VsVar     =  (          48 , CSHDR_EDIR )   'Important geometry: camera direction
   (…register 49 reserved for camera viewport size, set by standard camera …)
   #tag VsVar     =  (          50 , CSHDR_TIME )   'Application world time

##define (MSHD_STDPERPIXEL) as <MATERIAL_3A>
   #tag PsVar     =  (          1 , CSHDR_AMBI_A )
   #tag PsVar     =  (          2 , CSHDR_DIFF_A )
   #tag PsVar     =  (          3 , CSHDR_SPEC_A )
```

D103A – Unified GUI

- **SHD3D script also provides ready-to-use shaders for some classic, standard illumination models:**
  - **Standard Lambertian specular reflection vertex shader**
  - **Standard Blinn half-vector specular reflection vertex shader**
  - **Microsoft view space half-vector specular vertex shader**
  - **Standard Gouraud pixel shader**

```
##define (VS_STD_LREFL) as <VSHADER_3A>
   #tag File      =  "../Demo_Common/Files_Media/Shaders/Shader_Std_LambertRefl.vsh"
   #tag ShaderFcn =  "VS_Std_Lrefl"
   #tag ShaderAsm =  (           NO )
   #tag ShaderVs  =  (          1.1 )

##define (VS_STD_BHALF) as <VSHADER_3A>
   #tag File      =  "../Demo_Common/Files_Media/Shaders/Shader_Std_BlinnHalf.vsh"
   #tag ShaderFcn =  "VS_Std_Bhalf"
   #tag ShaderAsm =  (           NO )
   #tag ShaderVs  =  (          1.1 )

##define (VS_STD_MHALF) as <VSHADER_3A>
   #tag File      =  "../Demo_Common/Files_Media/Shaders/Shader_Std_MsoftHalf.vsh"
   #tag ShaderFcn =  "VS_Std_Mhalf"
   #tag ShaderAsm =  (           NO )
   #tag ShaderVs  =  (          1.1 )

##define (PS_STD_GOURAUD) as <PSHADER_3A>
   #tag File      =  "../Demo_Common/Files_Media/Shaders/Shader_Std_Gouraud.psh"
   #tag ShaderFcn =  "PS_Std_Gouraud"
   #tag ShaderAsm =  (           NO )
   #tag ShaderVs  =  (          1.2 )
```

D103A – Unified GUI

- **Finally, SHD3D provides a programmable kinex (Chapter 4) and animation controller to rotate displayed objects around the z-axis, with stop/start controlled by the UNV* buttons.**

```
##define (KX_AUTOROT) as <KINEX_I>
   #tag QLerpM    =  (    YES )
   #tag FrameK    =  (      0 ,  "Decl(M);RotZ(0.00);" )
   #tag FrameK    =  (      1 ,  "Decl(M);RotZ(90.0);" )
   #tag FrameK    =  (      2 ,  "Decl(M);RotZ(180.);" )
   #tag FrameK    =  (      3 ,  "Decl(M);RotZ(270.);" )
   #tag VsMat     =  (      0 ,   MAT_WT )
   #tag VsMat     =  (      8 ,  MAT_WVT )

##define (AN_AUTOROT) as <ANIMATOR_F>
   #tag Type      =  (     ANIM_SEQSTATE )
   #tag Clock     =  (         CLOCK_PMS )
   #tag AState    =  (     RN  ,       0 )
   #tag AState    =  (     RPA ,       0 )
   #tag AState    =  (     RPB ,       1 )
   #tag AState    =  (     RPC ,       2 )
   #tag AState    =  (     RPD ,       3 )
   #tag AState    =  (     RMA ,       0 )
   #tag AState    =  (     RMB ,       3 )
   #tag AState    =  (     RMC ,       2 )
   #tag AState    =  (     RMD ,       1 )
   #tag TransTMP  =  (     RPA ,     RPB ,      4000 )
   #tag TransTMP  =  (     RPB ,     RPC ,      4000 )
   #tag TransTMP  =  (     RPC ,     RPD ,      4000 )
   #tag TransTMP  =  (     RPD ,     RPA ,      4000 )
   #tag TransTMP  =  (     RMA ,     RMB ,      4000 )
   #tag TransTMP  =  (     RMB ,     RMC ,      4000 )
   #tag TransTMP  =  (     RMC ,     RMD ,      4000 )
   #tag TransTMP  =  (     RMD ,     RMA ,      4000 )
   #tag TransSIG  =  (     RN  ,     RPA ,    CH_NAV ,  X_UNVA )
   #tag TransSIG  =  (     RN  ,     RMA ,    CH_NAV ,  X_UNVB )
   #tag InterSIG  =  (     RN  ,    2000 ,    CH_NAV ,  X_UNVC )
```

D103A – Unified GUI

- **The SHD3D user code simply copies the control panel state variables into the float4 variables mapped to registers.**

```
void Common_ShaderConnect3D_Init (void* VArg)
BGN_ACTOR_CODE{

   /* Obtain pointers to the Principia objects defined in the script */
   Cs_LposA  = (SX_Variable*) Principia->GetReferenceTo("CSHDR_LPOS_A");
   Cs_LdirA  = (SX_Variable*) Principia->GetReferenceTo("CSHDR_LDIR_A");
   . . .

END_ACTOR_CODE}


void Common_ShaderConnect3D_Main (void* VArg)
BGN_ACTOR_CODE{

   /* Principal light: Copy panel control data to shader register variables. */
   . . .
   Cs_LparA->SetF->ePtr[0] = aL1->AttC0 ;
   Cs_LparA->SetF->ePtr[1] = aL1->AttC1 ;
   Cs_LparA->SetF->ePtr[2] = aL1->AttC2 ;
   Cs_LparA->SetF->ePtr[3] = aL1->Range ;
   . . .
```

- **It is that simple! However, if you write your own shaders to use with the rendering GUI, make sure your constant register indexes do not conflict with those used by the GUI.**


Chapter 1:03
Basic Rendering
Principia v3.0
Reference Series
(c) Western Star
Entertainment Ltd, 2003-2006

D103B – Geometry

- Most rendering is based on rasterizing 3D geometry primitives onto image buffers. Primitives are typically discrete approximations of object surfaces.
- The workhorse primitives for today's graphics are 3D triangle collections, commonly structured in vertex buffers.
- Chapter 3 focuses on basic management of 3D vertex buffers. Note that other frameworks and geometry types exist, and are supported by Principia.

D103B – Geometry

- **Basic Principia geometry components in the *3A* rendering framework:**
  - Vertex buffers (GX_VSet3A)
  - Kinexes (many types, but for now, we are interested only in the model matrix M of their AX_Kinex generic archetype)
- **Model vs. World Space:**
  - Vertex geometry data is typically specified in a coordinate system local to the discretized surface (model space)
  - Typically, this geometry needs to be transformed in the application world space by multiplication with the AX_Kinex::M model (or world) transform matrix.
  - Kinexes hold the model matrices applied when rendering vertex streams, and thus they define where in the 3D world will a vertex primitive be positioned and rendered.

D103B – Geometry

Spaces and transforms:

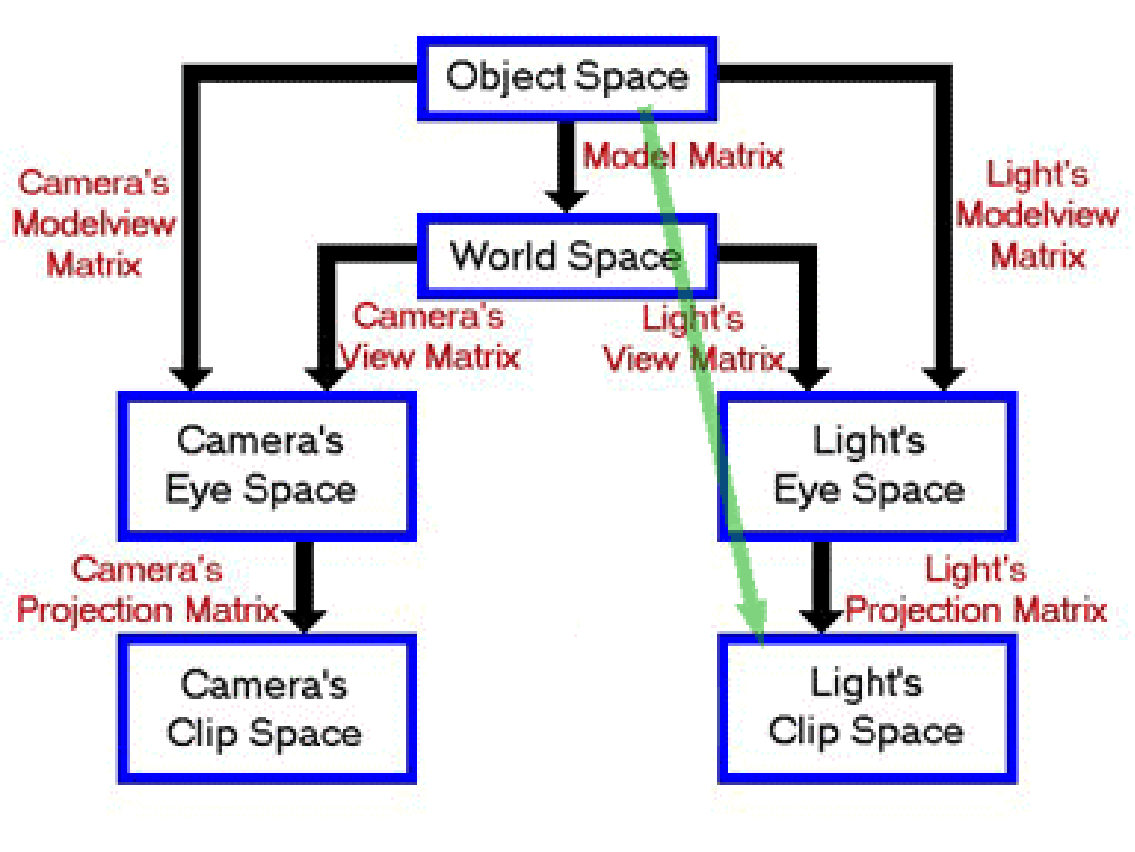
- Space: a coordinate system used to describe geometry data. Many “spaces” are used for rendering besides model and world spaces.
- Transform: an operation transforming coordinates from one space to another. Executed via a CPU or GPU MxV instruction.

Other commonly used spaces:

- View or camera space: Transform uses V matrix and outputs vertex position in view frustrum (extent is bound by camera settings).
- Clip space: Transform uses VxP matrix and ouputs vertex position in the [-1,1]xy+[0,1]z cube representing the projected view frustrum.
- Tangent space. Local system based upon the surface curvature.
- Light space: Coordinate system aligned with light source.
- UVW space: Coordinate system aligned with texture UV sets.
- Environment map space: Generic term for a transform from world space via some environment matrix, to create a coordinate system for sampling an environment map, such as a reflection map

D103B – Geometry

- **Some common spaces and transforms:**



D103B – Geometry

Transforms in DirectX

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix}$$

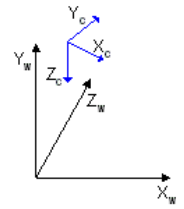
$$\begin{aligned} x' &= (x \times M_{11}) + (y \times M_{21}) + (z \times M_{31}) + (1 \times M_{41}) \\ y' &= (x \times M_{12}) + (y \times M_{22}) + (z \times M_{32}) + (1 \times M_{42}) \\ z' &= (x \times M_{13}) + (y \times M_{23}) + (z \times M_{33}) + (1 \times M_{43}) \end{aligned}$$

For efficiency reasons, Direct X executes the MxV multiply as

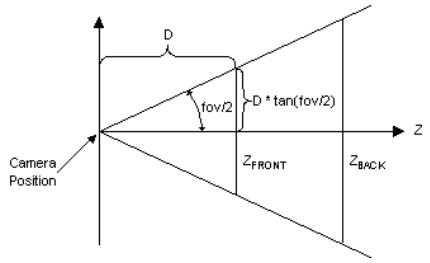
- [row vector] = [row vector] x [matrix]

D103B – Geometry

- **From world to screen …**
- **Camera space transform**
  - Translation and rotation
  - No scaling
- **Clip space transform (DX)**
  - Scaling of view frustrum
  - X,Y to [-1,1], Z to [0,1]
  - Vertex shader output
- **Viewport scale & center**
  - X,Y to screen
  - Z to depth



$$V = T \cdot R_z \cdot R_y \cdot R_x$$



$$\begin{bmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & Q & 1 \\ 0 & 0 & -QZ_n & 0 \end{bmatrix}$$

$$w = \cot\left(fov_w/2\right) \qquad w = \frac{2 \cdot Z_n}{V_w}$$

$$h = \cot\left(fov_h/2\right) \qquad h = \frac{2 \cdot Z_n}{V_h}$$

$$Q = \frac{Z_f}{Z_f - Z_n}$$

$$\begin{bmatrix} dwWidth/2 & 0 & 0 & 0 \\ 0 & -dwHeight/2 & 0 & 0 \\ 0 & 0 & dvMaxZ - dvMinZ & 0 \\ dwX + dwWidth/2 & dwHeight/2 + dwY & dvMinz & 1 \end{bmatrix}$$

D103B – Geometry Data

- **GX_VSet3A ! You will see it a lot:**
  - Workhorse vertex data primitive component in the Principia *3A* rendering framework. It encapsulates vertex buffers, many capabilities, and much else besides.
- **GX_VSet3A structure:**
  - Vertex data buffer
  - Optional index data buffer for indexed primitives
  - Definer (Principia DX_VtxDef3 component)
  - Topology type and keyframe boundaries data
  - Optional embedded material
  - Optional embedded extra GX_VSet3A includes
  - Optional embedded geometry phase GX_VSet3A includes
  - Geometry bounding box, sphere...etc. in model space

D103B – Geometry Data

- **GX_VSet3A capabilities**
  - Balance between flexibility and overhead
  - Load from multiple file formats
  - Save into 3DS and native VSF file formats
  - Usable with shaders and/or the fixed function pipeline
  - Hierarchical flexibility: embeds materials and other GX_VSet3A internally, while serves as component in renderable meshes.
  - Configuration flexibility: all aspects of format, topology, usage options, structure ...etc. are managed from script.
  - User choice of indexed or non-indexed topology
  - Stream definition and usage driven by simple script language
  - Wide variety of effects, topologic operations and transformations
  - Native animation support featuring discrete keyframes
  - Multiple access pathways to vertex and index data
  - Alternate vertex data selection based on LoD controllers

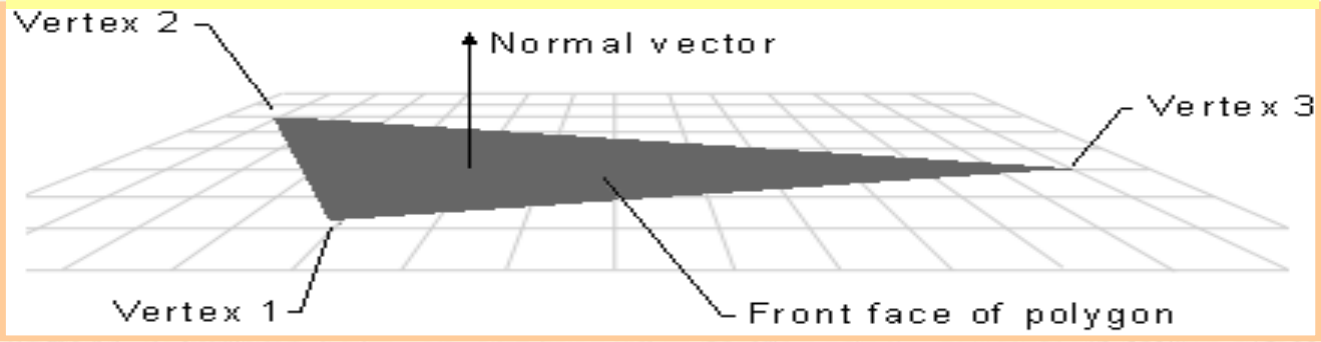
D103B – Geometry Data

- **Vertex data buffers:**
  - Series of identical vertices that are rendered in one pass by the GPU, subject to current material-kinex-camera states.
  - Vertex format: provided in the definer, describes the type of data each vertex contains (e.g. position, normals ...etc).
  - Geometry type: defines how vertices are topologically connected to form triangles outlining the shape rendered.
  - Vertex data elements are treated as separate but interleaved binary streams by the GPU.
- **Vertex data geometry types for GX_VSet3A**
  - Point list
  - Line list
  - Line strip
  - Triangle list
  - Triangle strip
  - Triangle fan

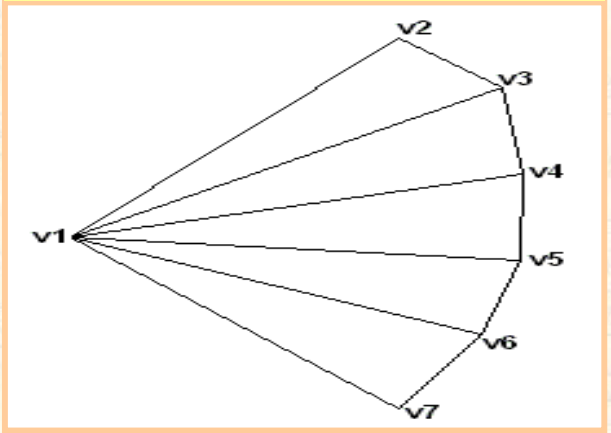
D103B – Geometry Data

- **Vertex buffer data topology conventions. Non-indexed buffer vertices are rendered in the sequence they appear.**

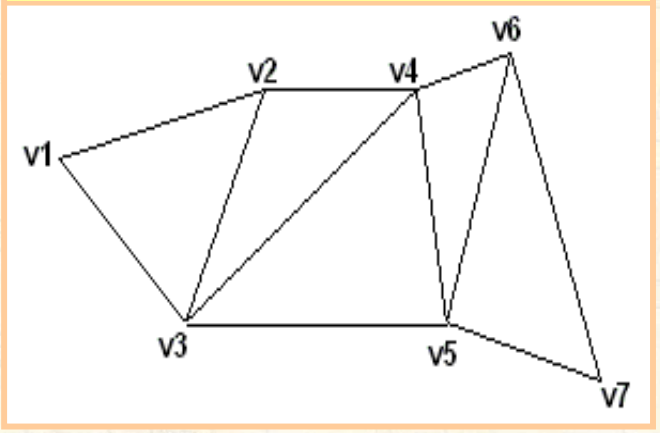
Principia/DX9 front face convention for single triangle



Simple triangle fan topology



Simple triangle strip topology



D103B – Geometry Data

Index buffers

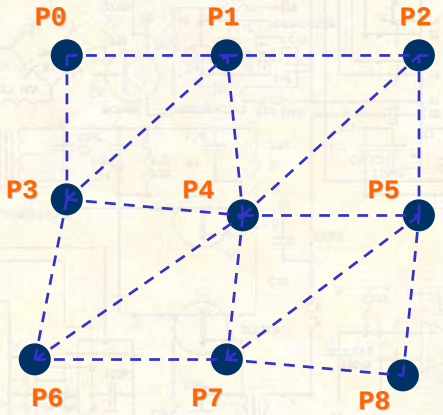
- Without index buffers, the vertex buffer needs to encode the triangle topology (by listing the vertices in the right order) in addition to storing vertex data. This requires the redundant storage of vertices.
- Index buffers store the indexes of vertices from the vertex buffer in the order they form triangles. The vertices do not need to be stored redundantly.

Indexed buffer formats

- IsIndexed flag and index buffer format declared in the DX_VtxDef3() definer.
- Format is either 16 or 32 bit. Some older or mobile graphic cards do not support 32-bit format, which restricts the count of vertices to less than 65,355.

D103B – Geometry Data

- Indexed vertex set data organization (example)



  - Zero-based indexing
  - Vertex list: P0..P8
  - Index for triangle set topology is (0,1,3), (1,4,3), (1,2,4), (2,5,4), (3,4,6), (4,7,6), (4,5,7), (5,8,7)... (n,n+1,n+2),(n+1,n+3,n+2)

- A non-indexed vertex set would have to specify all listed vertices above in the vertex buffer instead of just the indices. Contrast this to an indexed strip (bandwidth reduced by 3x!)

D103B – Geometry Data

- **Much of the geometry used in production applications is based on indexed buffers:**
  - Since a vertex typically takes far less space than a 16/32-bit index, considerable GPU memory savings are achieved relative to non-indexed buffers.
  - Rendering and loading bandwidth is improved.
  - Easier loading and manipulation in many cases.
- **Index buffers have some limitations too:**
  - Objects with sharp edges, where the same vertex may have multiple normals, are harder to manage.
  - Some effects which rely on face disjunctions, such as shattering...etc, require vertex redundancy.

D103B – Geometry Data

- **Principia makes the management of indexed and non-indexed vertex data transparent and easy for the user:**
  - The VSet* family of Principia components holds the vertex data as either indexed or not, and tracks the topology automatically.
  - VSet* 3DS loaders and most procedural geometry generators enable the user to choose between indexed topology or not by means of a simple flag.
  - VSet* definers enable separate driver properties for index and vertex data. Principia provides DMA methods for accessing index as well as vertex data.
- **Unless designing custom effects or geometries, Principia users do not need to delve into indexing details. Just set the “indexed” flag!**

D103B – Geometry Data

- **Where does vertex data come from ?**
  - Loaded from file in the native Principia VSF3 format. Such files usually hold a binary image of the definer and data buffers, and load the fastest.
  - Loaded from file in external formats supported by Principia (e.g. 3DS or OBJ). Note that complex objects will load, but their data structure will be hard to manage and may degrade performance.
  - Created from one of the many Principia procedures that generate or modify geometry data.
  - Created in the user code by taking the data from internal Principia components such as SX_MeshUTL.
  - Created in the user code totally from scratch.
- **Because VSF3 files do not force mesh and material structures on geometry, they are very suitable for high-performance productions.**

D103B – Geometry Data

The essential DX_VtxDef3() definer:

- Principia associates each vertex stream with a DX_VtxDef3() object that holds key information on stream properties and vertex format.
- Amongst others, the VtxDef3 object carries a flag to signal the GPU that the stream is to be rendered with the legacy fixed function pipeline. Leave this flag alone if using shaders.

VtxDef3() specification mechanisms:

- *Explicit definition in script.* The definition can be (a) discrete flags or (b) enumerated DECL statements. Pre-DX9 FVF streams must use discrete flags. DECL statements allow much richer designs.
- *From file.* Vertex streams loaded from a VSF file inherit the format stored in the preamble of the vertex stream data. Vertex stream imported from 3DS or other formats fall in the following category.
- *Implicit initialization upon creating vertex streams.* In this case, the format is inferred from the creation context (e.g. structure of internal SX_MeshUTL component) or from format hints/arguments used by the vertex stream constructors.

D103B – Geometry Data

- **Vertex format text mode descriptor format**
  - Usage:Type:[stream:usage index]:[method]
  - Coma-separated code sequence describing stream
  - Case-insensitive
- **Usage codes, required:**
  - POS: untransformed position
  - PIX: transformed position
  - NORM: normal vector
  - TAN: tangent vector
  - BINORM: binormal vector
  - COLOR: color data
  - CD,CS: specific diffuse or specular color data
  - TEX: texture coordinates
  - WEIGHT: vertex skinning weight data
  - INDEX: vertex skinning index data
  - POINT: sprite point field

D103B – Geometry Data

- **Type codes, required:**
  - 1F..4F : one to four 32-bit floats
  - COLOR: four unsigned bytes packed in DWORD
  - U: four unsigned bytes packed in an index
  - U4N: four unsigned bytes pack normalized by 255.0f
  - S2,S4: two/four component signed SHORT packed in index
  - S2N,S4N: two/four component signed SHORT normalized by 32767.0f
  - SU2N,SU4N: two/four component unsigned SHORT normalized by 65525.0f
  - F162,F164: two or four-component 16-bit float
  - UDEC3, DEC3N: three component 10-10-10bit data
- **Stream:UsageIndex codes, optional**
  - Usage indexes indicate special processing by the vertex unit. Refer to the DirectX documentation for the values that can be used.
- **Method code, optional**
  - Controls tesselation and displacement mapping processing by the vertex unit. Refer to the DirectX documentation for the values that can be used.

D103B – Geometry Data

- **DX_VtxDef3() definer contents**
  - Human-readable vertex/index buffer format specs.
  - GPU-translated vertex/index buffer formats
  - Vertex buffer memory pool type
  - Vertex buffer usage codes
  - Vertex buffer locking codes for DMA access
  - Index buffer memory pool type
  - Index buffer usage codes
  - Index buffer locking codes for DMA access
  - Vertex and index elements unit memory size
  - Vertex and index elements memory footprint

D103B – Geometry Data

- The geometric placement and appearance of vertex data on screen is mainly controlled by:
  - Kinexes (world transform, skinning, frame states)
  - Camera view definition (introduced in D103D)
  - GPU render states
- Geometry-controlling GPU render states (many states unique to the fixed-function pipeline are not listed)
  - RS_ZENABLE, RS_ZWRITEENABLE + all depthstencil RS
  - RS_FILLMODE
  - RS_SHADEMODE
  - RS_CULLMODE
  - RS_CLIPPING
  - RS_LOCALVIEWER
  - RS_NORMALIZENORMALS
  - RS_CLIPPLANEENABLE + all user clip plane settings

D103B – Kinexes

- Kinexes are essential Principia components that hold data describing the state of an object. All object renders are accompanies by a kinex.
- Principia provides many kinexes to power a wide range of animation, skinning and special effects.
- All kinexes hold a world (or model) transform matrix that transforms vertex coordinates from object model space to world, and thereby positions, rotates and distorts geometry.
- This chapter shows how to set and use this matrix to control rendered primitive geometry.

D103B – Kinexes

- **All kinexes share the following data, referencing the current state of the rendered geometry**
  - **M: World transform matrix (of interest here)**
  - **B[]: Set of vertex blend (skinning) matrices, optional**
  - **T[]: Set of texcoord/general use matrices, optional**
  - **Frame: Animation keyframe index**
  - **Value: Animation keyframe interpolation position**
  - **Timer: Animation timer index**
  - **State/Locus: Animation state references**
  - **Material: Optional material reference. Should only be used to map transform matrices to shader registers.**

D103B – Kinex World Matrix

- **World (model) transform matrix KX->M**
  - Determines object position in the world by standard 3D MxV operation applied to object root coordinates in model space.
  - The object can be placed anywhere, rotated and distorted at will within the limits of affine transformations.
- **Access to any kinex model matrix "M":**
  - Script (RO): multiple methods of setting matrix available
  - Code (RW): Kinex->M is world transform 4x4 matrix object
- **Accessing 3D object model matrices:**
  - 3D objects instantiate their own internal kinex by copy from the template kinex provided in the script.
  - To access the object model matrix, reference the object and then access its kinex model matrix using ::GetPosition(). Do not access the object template kinex used in script directly.

D103B – Kinex Syntax (1)

- **Model matrix script specification methods**
  - Direct specification of world transform 4x4 matrix
  - Sequential product of transforms
  - Combination of rotation, translation and scaling
  - Pseudo-linguistic shorthand (KPS) specification
- **Tags for direct specification:**
  - <Row1>: World transform matrix row 1 (M11..M14)…
  - <Row4>: World transform matrix row 4 (M41..M44)
  - Follows Microsoft® DirectX® convention
  - M41..M43 are translation/positioning coordinates
- **Tags for sequential product specification:**
  - <Trans1..5>: Sequence of up to five model kinexes
  - The resulting matrix is M = M1 x M2 x M3 x M4 x M5
  - Follows Microsoft® DirectX® convention for the MxM order of sequenced transforms (M1 is the first transform…etc).

D103B – Kinex Syntax (2)

- **Tags for specifying a combination of affine transforms:**
  - TrsVec: X,Y,Z translation components
  - SclCtr: X,Y,Z of world transform scaling center
  - SclAmt: X,Y,Z scaling factors
  - RotCtr: X,Y,Z of rotation axis center point
  - RotAxe: Unit vector X,Y,Z along rotation axis
  - RotAmt: Rotation amount in degrees
- **Special kinexes**
  - A kinex with a unit world transform matrix takes the object raw coordinates in model space as they are, and uses them as world coordinates for viewing.
  - A NULL kinex argument for most operations requiring a kinex means to use the previous world transform state.

D103B – Kinex Syntax (3)

- **Kinex pseudolinguistic shorthand specification (KPS)**
  - Shorthand language-like string description of the most common transforms that make up the kinex matrices. While not as general as the typed kinexes, KPS is much shorter, easier to read, and suitable for production scripts.
- **KPS usage**
  - When specifying simple kinexes, KPS string on the definition line, instead of tag list below.
  - Whenever expecting an argument kinex by reference, enter KPS string instead of kinex.
- **KPS syntax**
  - "Token;Token;Token;..." ;-delimited sequence
  - "Token" = Cmd(Arg,Arg...)
  - "Cmd" is a transform or definition command
  - Transforms are chained from left (first) to right (last)

D103B – Kinex Syntax (4)

- **KPS commands**
  - Decl(*): specifies which kinex matrix will be described
  - Rot(a,x,y,z): rotate by <a> around axis (x,y,z)
  - Rotx(a)...Roty(a): rotate by <a> around respective axis
  - Ypr(y,p,r): Yaw-pitch-roll rotation. All angles are in degrees.
  - Mul(x,y,z): scale around origin by <x,y,z> factors
  - Mov(x,y,z): translate by (x,y,z)
  - Set(Code,Index): sets the kinex matrix denoted by <code> at shader register <Index>. Essential for shader rendering.
- **Matrix declarations**
  - If no declaration present, subsequent commands are assumed to reference the base kinex matrix M.
  - If declaration present, all commands refer to the matrix referenced in the declaration
  - Available declarations: M, B1..B3, T0..T7 for respective base, blend and texture coordinate/general transform matrices.

D103B – Kinex Syntax (5)

- **Principia rotation conventions**
  - Many Principia components require the specification of multiple rotations (e.g. bone animation keyframes).
  - There are many conventions to orient and sequence multiple rotation angles. Principia uses the two most standard formulations: quaternion and ETB-YPR.
- **Quaternion rotation**
  - Best practice. Rotation data internally derived by specifying CCW rotation angle about an axis vector Rot(a,x,y,z).
- **Euler Tait-Bryan angles Yaw-Pitch-Roll (ETB-YPR) rotation**
  - Common aeronautical practice. Do not confuse with the other dozen YPR conventions (such as the one in DirectX).
  - First rotation: Roll, about (body) x-axis
  - Second rotation: Pitch, about (body) y-axis
  - Third rotation: Yaw, about (body) z-axis
  - Arguments in script entered in reverse Ypr(yaw,pitch,roll)

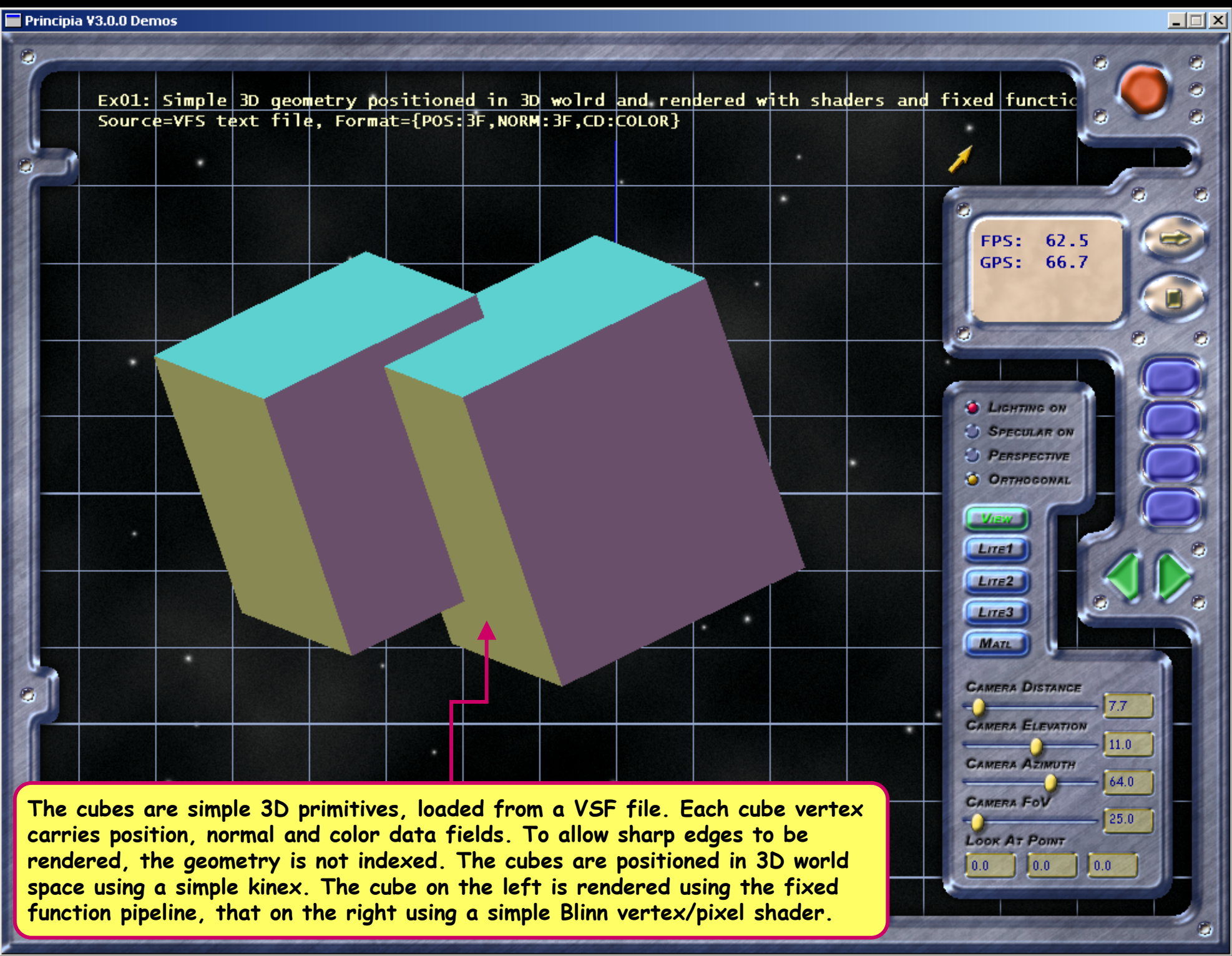
D103B – Kinex Syntax (6)

- **KPS example: The non-KPS definition of these simple and keyframed kinexes would run over 200 script lines, and be much harder to read.**

```
##define (KX_BODY_EX05) as <KINEX_I>
   #tag QLerpM    =  (    YES )
   #tag FrameK    =  (      0 ,  "Decl(M);Rotz(-10);" )
   #tag FrameK    =  (      1 ,  "Decl(M);Rotz(0.0);" )
   #tag FrameK    =  (      2 ,  "Decl(M);Rotz(10.);" )
   #tag FrameK    =  (      3 ,  "Decl(M);Rotz(0.0);" )

##define <KINEX_S>  = ( KX_HIPPL_EX05,  "Decl(M);Rot(60,0,1,0);Mov(1,0,0);"  )
##define <KINEX_S>  = ( KX_KNEEL_EX05,  "Decl(M);Rot(30,0,1,0);Mov(1,0,0);"  )

##define (KX_MEM1L_EX05) as <KINEX_I>
   #tag QLerpM    =  (    YES )
   #tag FrameK    =  (      0 ,  "Decl(M);Rot(-10,0,0,1)" )
   #tag FrameK    =  (      1 ,  "Decl(M);Rot(0,0,0,1)"   )
   #tag FrameK    =  (      2 ,  "Decl(M);Rot(50,0,0,1)"  )
   #tag FrameK    =  (      3 ,  "Decl(M);Rot(0,0,0,1)"   )

##define (KX_MEM2L_EX05) as <KINEX_I>
   #tag QLerpM    =  (    YES )
   #tag FrameK    =  (      0 ,  "Decl(M);Rot(10,0,0,1)"  )
   #tag FrameK    =  (      1 ,  "Decl(M);Rot(0,0,0,1)"   )
   #tag FrameK    =  (      2 ,  "Decl(M);Rot(-40,0,0,1)" )
   #tag FrameK    =  (      3 ,  "Decl(M);Rot(0,0,0,1)"   )

##define <KINEX_S>  = ( KX_HIPPR_EX05,  "Decl(M);Rot(120,0,1,0);Mov(-1,0,0);"  )
##define <KINEX_S>  = ( KX_KNEER_EX05,  "Decl(M);Rot(-30,0,1,0);Mov(1,0,0);"  )

##define (KX_MEM1R_EX05) as <KINEX_I>
   #tag QLerpM    =  (    YES )
   #tag FrameK    =  (      0 ,  "Decl(M);Rot(50,0,0,1)"   )
   #tag FrameK    =  (      1 ,  "Decl(M);Rot(0,0,0,1)"    )
   #tag FrameK    =  (      2 ,  "Decl(M);Rot(-10,0,0,1)"  )
   #tag FrameK    =  (      3 ,  "Decl(M);Rot(0,0,0,1)"    )

##define (KX_MEM2R_EX05) as <KINEX_I>
   #tag QLerpM    =  (    YES )
   #tag FrameK    =  (      0 ,  "Decl(M);Rot(-40,0,0,1)"   )
   #tag FrameK    =  (      1 ,  "Decl(M);Rot(0,0,0,1)"    )
   #tag FrameK    =  (      2 ,  "Decl(M);Rot(10,0,0,1)"  )
   #tag FrameK    =  (      3 ,  "Decl(M);Rot(0,0,0,1)"    )
```

D103B – Concept Outline

- **D103B starts with some very simple basics. This is the gist of topics covered in this demo:**
  - **Understanding how to position geometry, and how to create and read VFS3 file data.**
  - **Accessing and creating geometry data from within the user code.**
  - **Importing geometry data from 3DS formats, and understanding how to best import and use complex multi-mesh, multi-material objects.**
  - **Advanced topics such as streaming, skinning or procedural generation are covered separately.**
- **Despite being called geometry primitives, vertex streams may carry data that has little to do with geometry (e.g. color, occlusion factors).**


Principia V3.0.0 Demos
Ex01: Simple 3D geometry positioned in 3D wolrd and rendered with shaders and fixed functio
Source=VFS text file, Format={POS:3F,NORM:3F,CD:COLOR}
FPS: 62.5
GPS: 66.7
Lighting on
Specular on
Perspective
Orthogonal
View
Lite1
Lite2
Lite3
Matl
Camera Distance
7.7
Camera Elevation
11.0
Camera Azimuth
64.0
Camera FoV
25.0
Look At Point
0.0
0.0
0.0
The cubes are simple 3D primitives, loaded from a VSF file. Each cube vertex carries position, normal and color data fields. To allow sharp edges to be rendered, the geometry is not indexed. The cubes are positioned in 3D world space using a simple kinex. The cube on the left is rendered using the fixed function pipeline, that on the right using a simple Blinn vertex/pixel shader.

D103B:Ex01 - Geometry

Objectives:

- Show geometry and vertex data basics

Requirements and Implementation:

- Manually create a cube geometry in a FVS format and load it into a Principia vertex stream component
- Render two instances of this vertex stream, one using the fixed function pipeline, one using basic shaders.
- Position the two instances at different locations, using two different scripted kinexes.

Notes:

- Following Ex01, the use of the fixed function pipeline will be limited to the simplest cases, and will not be highlighted. Refer to Chapter 11 to learn more about this legacy rendering method (and some basics taken for granted here).
- The screen and depth buffers used to render are created during the graphic interface creation in the script include.

D103B:Ex01 - Geometry

Fixed function cube implementation:

- Define a material with the proper render states
- Load the geometry from data. Use a definer that specifies to use a FVF-style vertex format (required for fixed function rendering).
- Encapsulate the geometry into a mesh (material+geometry), which is combined into an object (kinex+mesh). Use the kinex to place the object where it is wanted, using the KPS scripting described earlier.
- The object is added to a world that is part of a viewer control which renders the world as part of the scene controls sequence.

```
##define (M_EX01_FIX) as <MATERIAL_3A>
   #tag RS        =  (               RS_FILLMODE ,        RA_SOLID )
   #tag RS        =  (               RS_CULLMODE ,      RA_CULLCCW )
   #tag TS        =  (                 TS_COLOROP ,   TO_SELECTARG2 ,   0 )
   #tag TS        =  (               TS_COLORARG1 ,      TA_TEXTURE ,   0 )
   #tag TS        =  (               TS_COLORARG2 ,      TA_DIFFUSE ,   0 )
   #tag TS        =  (                 TS_ALPHAOP ,   TO_SELECTARG2 ,   0 )
   #tag TS        =  (               TS_ALPHAARG1 ,      TA_TEXTURE ,   0 )
   #tag TS        =  (               TS_ALPHAARG2 ,      TA_DIFFUSE ,   0 )
   #tag VShader   =  (                      NONE )
   #tag PShader   =  (                      NONE )

##define (G_EX01_FIX) as <VERTEXSET_3A>
   #tag File      =  "Files_Media\Mesh_UnitCube_RND.vsf"
   #tag Definer   =  ( VTD_FVF_FILE )
   #tag LoadNow   =  (            1 )

##define (R_EX01_FIX) as <MESH_3A>
   #tag Component =  (            0 ,      M_EX01_FIX ,   G_EX01_FIX )

##define (O_EX01_FIX) as <GENOBJECT_3A>
   #tag Construct =  ( "Decl(M);Mul(0.7,0.7,0.8);Rot(-20,0,-45);Mov(0.5,0.0,0.0);" ,      R_EX01_FIX ,     NONE )
```

D103B:Ex01 - Geometry

Shader-based cube implementation:

- Define the shaders and reference them in the material used. Remember to include the material from the NAV3D script that maps shader constant registers to the light and camera parameters (to enable our control panel).
- Load the geometry from data. Use a standard definer. From this point on, everything else is the same as with the fixed-function cube.

```
##define (VS_EX01_BLINN) as <VSHADER_3A>
   #tag File      =  "Ex01_VS_Blinn.shd"
   #tag ShaderFcn =  "Ex01_VS_Blinn"
   #tag ShaderAsm =  (           NO )
   #tag ShaderVs  =  (          1.1 )

##define (PS_EX01_DIFUS) as <PSHADER_3A>
   #tag File      =  "Ex01_PS_Diffuse.shd"
   #tag ShaderFcn =  "Ex01_PS_Diffuse"
   #tag ShaderAsm =  (           NO )
   #tag ShaderVs  =  (          1.2 )

##define (M_EX01_SHD) as <MATERIAL_3A>
   #tag RS        =  (              RS_FILLMODE ,        RA_SOLID )
   #tag RS        =  (              RS_CULLMODE ,      RA_CULLCCW )
   #tag VShader   =  (            VS_EX01_BLINN )
   #tag PShader   =  (            PS_EX01_DIFUS )
   #tag Material  =  (       MSHD_STDCAMLIT , - )

##define (G_EX01_SHD) as <VERTEXSET_3A>
   #tag File      =  "Files_Media\Mesh_UnitCube_RND.vsf"
   #tag Definer   =  ( VTD_DCL_FILE )
   #tag LoadNow   =  (           1 )

##define (R_EX01_SHD) as <MESH_3A>
   #tag Component =  (           0 ,      M_EX01_SHD ,   G_EX01_SHD )

##define (O_EX01_SHD) as <GENOBJECT_3A>
   #tag Construct =  ( "Decl(M);Mul(0.8,0.7,0.9);Rot(-20,0,-45);Mov(0.0,0.5,0.0);Set(tM,0);" ,      R_EX01_SHD ,     NONE )
```

D103B:Ex01 - Geometry

What is in the VSF file that holds the vertex data?

```
TEXT
VS3A
TRIANGLE LIST PRIMITIVE 3D OBJECT
@@ Vertex format specification
POS:3F, NORM:3F, COLOR:COLOR
@@ Primitive type
4
@@ Indexed flag
0
@@ Reserved bootstrap section
0
@@ Local material code
0
@@ Total number of vertices and indices
36
0
@@ Number of frames
1
@@ Frame#0: Primitive first vertex index, primitive count, number of vertices
0  12  36
@@ Frame#0: Locus of first frame in index array and index count per frame
0  0
@@ Vertex data Tr01
1.00  1.00  1.00   0.0  0.0  1.0   AAFFFFFF
1.00  0.00  1.00   0.0  0.0  1.0   AAFFFFFF
0.00  0.00  1.00   0.0  0.0  1.0   AAFFFFFF
@@ Vertex data Tr02
1.00  1.00  1.00   0.0  0.0  1.0   AAFFFFFF
0.00  0.00  1.00   0.0  0.0  1.0   AAFFFFFF
0.00  1.00  1.00   0.0  0.0  1.0   AAFFFFFF
```

```
@@ Vertex data Tr03
0.00  1.00  1.00   0.0  1.0  0.0   AAFFFFFF
0.00  1.00  0.00   0.0  1.0  0.0   AAFFFFFF
1.00  1.00  1.00   0.0  1.0  0.0   AAFFFFFF
@@ Vertex data Tr04
0.00  1.00  0.00   0.0  1.0  0.0   AAFFFFFF
1.00  1.00  0.00   0.0  1.0  0.0   AAFFFFFF
1.00  1.00  1.00   0.0  1.0  0.0   AAFFFFFF
@@ Vertex data Tr05
1.00  0.00  0.00   1.0  0.0  0.0   AAFFFFFF
1.00  1.00  1.00   1.0  0.0  0.0   AAFFFFFF
1.00  1.00  0.00   1.0  0.0  0.0   AAFFFFFF
@@ Vertex data Tr06
1.00  0.00  0.00   1.0  0.0  0.0   AAFFFFFF
1.00  0.00  1.00   1.0  0.0  0.0   AAFFFFFF
1.00  1.00  1.00   1.0  0.0  0.0   AAFFFFFF
@@ Vertex data Tr07
0.00  1.00  1.00   -1.0  0.0  0.0   AAFFFFFF
0.00  0.00  1.00   -1.0  0.0  0.0   AAFFFFFF
0.00  0.00  0.00   -1.0  0.0  0.0   AAFFFFFF
@@ Vertex data Tr08
0.00  1.00  1.00   -1.0  0.0  0.0   AAFFFFFF
0.00  0.00  0.00   -1.0  0.0  0.0   AAFFFFFF
0.00  1.00  0.00   -1.0  0.0  0.0   AAFFFFFF
@@ Vertex data Tr09
0.00  1.00  0.00   0.0  0.0  -1.0   AAFFFFFF
0.00  0.00  0.00   0.0  0.0  -1.0   AAFFFFFF
1.00  1.00  0.00   0.0  0.0  -1.0   AAFFFFFF
@@ Vertex data Tr10
0.00  0.00  0.00   0.0  0.0  -1.0   AAFFFFFF
1.00  0.00  0.00   0.0  0.0  -1.0   AAFFFFFF
1.00  1.00  0.00   0.0  0.0  -1.0   AAFFFFFF
@@ Vertex data Tr11
0.00  0.00  0.00   0.0  -1.0  0.0   AAFFFFFF
1.00  0.00  1.00   0.0  -1.0  0.0   AAFFFFFF
1.00  0.00  0.00   0.0  -1.0  0.0   AAFFFFFF
@@ Vertex data Tr12
0.00  0.00  0.00   0.0  -1.0  0.0   AAFFFFFF
0.00  0.00  1.00   0.0  -1.0  0.0   AAFFFFFF
1.00  0.00  1.00   0.0  -1.0  0.0   AAFFFFFF
```

D103B – VSF File Structure (1)

- **Standard content management flags**
  - 3 items (TEXT/BINF, VSF3, name)
- **Vertex format descriptors (VtxDef structure block)**
  - Usage:Type:[stream:usage index]:[method]
  - Coma-separated code sequence describing stream
  - Case-insensitive
  - For fixed pipeline rendering, the format descriptors must translate to a valid FVF format (e.g. cannot have multiple positions)
- **Primitive type/topology specifier**
  - 1=point list
  - 2=line list, 3=line strip
  - 4=triangle list, 5=triangle strip, 6=triangle fan
- **Indexed flag.**
  - No indexed geometry used – we want sharp edges everywhere.
- **Reserved code for advanced data**
  - Not discussed here, 0 if no socket is present

D103B – VSF File Structure (2)

- **Number of frames**
  - Instances of vertex set in different animation configurations)
- **Frame data block (for all frames)**
  - Primitive first vertex index, primitive count, number of vertices
  - Locus of first frame in index array and index count per frame
- **Vertex data block (repeated per frame)**
  - Vertex data stream (1 vertex per line for text files)
- **Index data block (repeated per frame)**
  - Primitive forming indices into vertex array (1 element per line)
- **Note that materials data or embedded geometry sub-components are not embedded in Principia VSF files.**
  - Principia favors flexibility and separates the material from the geometry rendered. Principia has separate material read procedures.

D103B:Ex01 - Geometry

Geometry and Shaders

- The vertex shader declaration section states which constant registers hold the various transform matrices (and other environment data).
- The vertex shader declaration also describes the vertex format. This should match what is in the definer (although most hardware would forgive not declaring elements present in the stream).

```
(. . . The register bindings are from VS_Std_Include.shd common include file . . .)
float4x4 mWrld     : register (c0);   // Api: world matrix
float4x4 mViewProj : register (c4);   // Api: VxP   matrix
float4x4 mWrldView : register (c8);   // Api: WxV   matrix
float4x4 mView     : register (c12);  // Api: view  matrix
float4x4 mProj     : register (c16);  // Api: proj  matrix
float3   vLDir     : register (c31);  // Api: light dir
float4   vLAmbi    : register (c33);  // Api: ambient color
float4   vLDiff    : register (c34);  // Api: diffuse color
float4   vLSpec    : register (c35);  // Api: spcular color
float4   vMPwr     : register (c46);  // Api: matl power
float3   vEPos     : register (c47);  // Api: eye pos
float3   vEDir     : register (c48);  // Api: eye dir
float3   vPort     : register (c49);  // Api: viewport size
float3   vTime     : register (c50);  // Api: world timer


struct VS_INPUT
{   // Inbound vertex buffer stream
    float3 vPosition        : POSITION ;
    float3 vNormal          : NORMAL   ;
    float4 vDiffuse         : COLOR0   ;
};

struct VS_OUTPUT
{   // Extant processed vertex stream
    float4 vPosition        : POSITION ;
    float4 vDiffuse         : COLOR0   ;
    float4 vSpecular        : COLOR1   ;
};
```

D103B:Ex01 - Geometry

Geometry and Shaders

- The vertex shader program generates transformed vertex coordinates in screen space to pass on to the rasterizer. It also determines the lit vertex color in 3D space. These tasks are all geometry-driven, and will be detailed in later demos dedicated to shader design.

```
VS_OUTPUT Ex01_VS_Blinn(const VS_INPUT In)
{   VS_OUTPUT Out     = (VS_OUTPUT) 0;

    /* Position world-view-projection transform and position output */
    wPos              = mul(float4(In.vPosition,1.0f), mWrld);
    Out.vPosition     = mul(wPos, mViewProj);

    /* Normals world transform */
    wNorm             = mul(In.vNormal, (float3x3) mWrld);
    wNorm             = normalize(wNorm);

    /* Common light, view and half vectors in world space */
    wLite             = normalize(-vLDir);    // Lite pos - Vertex pos
    wView             = vEPos - wPos.xyz;     // View pos - Vertex pos
    wView             = normalize(wView);
    wHalf             = 0.5*(wLite+wView);
    wHalf             = normalize(wHalf);

    /* Lambert diffuse and ambient per-vertex color output */
    LdotN             = saturate(dot(wNorm, wLite));
    Out.vDiffuse.rgb  = (vLDiff*LdotN*In.vDiffuse.rgb + vLAmbi);
    Out.vDiffuse.a    = float1(1.0);

    /* Blinn specular per-vertex color output */
    HdotN             = saturate(dot(wNorm, wHalf));
    Out.vSpecular.rgb = vLSpec*pow(HdotN, vMPwr);
    Out.vSpecular.a   = float1(1.0);

    /* Output stream to pixel shader */
    return Out;
}
```

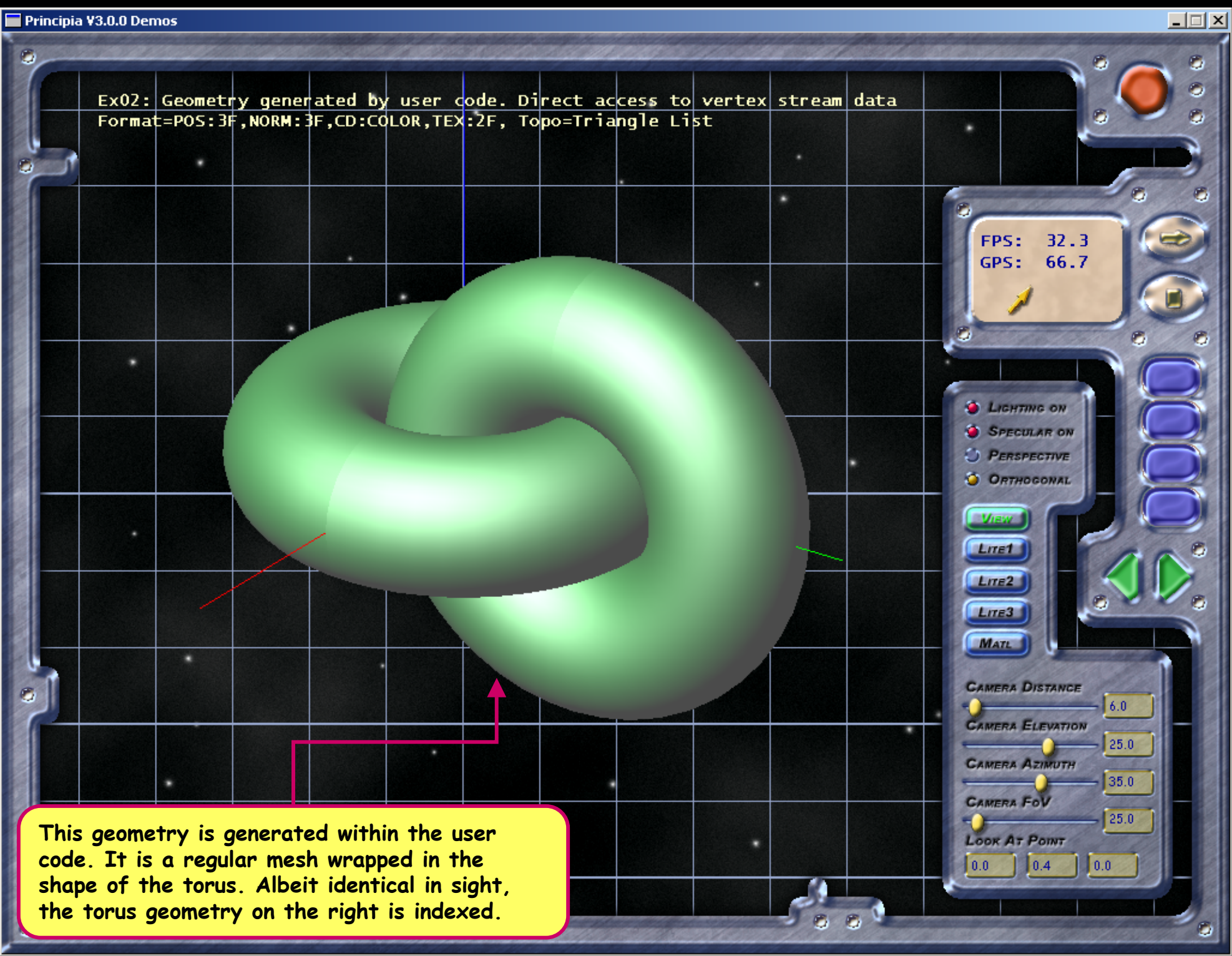
D103B:Ex01 - Geometry

What is in our two definers?

- When Principia reads geometry from a file, the definer format is overridden by the vertex format that is described in the file data.
- Note that each definer has different FVF flag. Programmable-based shader rendering is the standard today, but FVF renders can be useful at times.

```
##define (VTD_DCL_FILE) as <DC_VTXDEF3>
   #tag IsFVF     =  (            NO )
   #tag Format    =  "POS:3F, NORM:3F, TEX:2F"
   #tag VtxMemory =  (   GMEM_DEVICE )
   #tag IdxMemory =  (   GMEM_DEVICE )
   #tag IndexFmt  =  (       INDEX16 )
   #tag IsDynamic =  (             - )
   #tag IsWrtonly =  (             - )

##define (VTD_FVF_FILE) as <DC_VTXDEF3>
   #tag IsFVF     =  (           YES )
   #tag Format    =  "POS:3F, NORM:3F, TEX:2F"
   #tag VtxMemory =  (   GMEM_DEVICE )
   #tag IdxMemory =  (   GMEM_DEVICE )
   #tag IndexFmt  =  (       INDEX16 )
   #tag IsDynamic =  (             - )
   #tag IsWrtonly =  (             - )
```

We described what we wanted to render in script, and let Principia do it. It is exactly the same idea for much more complex renders – just materials, objects and worlds require more script, not code!


Principia V3.0.0 Demos
Ex02: Geometry generated by user code. Direct access to vertex stream data
Format=POS:3F,NORM:3F,CD:COLOR,TEX:2F, Topo=Triangle List
FPS: 32.3
GPS: 66.7
Lighting on
Specular on
Perspective
Orthogonal
View
Lite1
Lite2
Lite3
Matl
Camera Distance
6.0
Camera Elevation
25.0
Camera Azimuth
35.0
Camera FoV
25.0
Look At Point
0.0
0.4
0.0
This geometry is generated within the user code. It is a regular mesh wrapped in the shape of the torus. Albeit identical in sight, the torus geometry on the right is indexed.

D103B:Ex02 - Geometry

Objectives

- Show how to access geometry data from the user code for both indexed and non-indexed vertex sets.

Requirements and Implementation

- Define object and vertex geometry in script. Do not load the vertex data however, leave it all blank.
- Create the geometry in the user code (torus) by writing directly into the vertex set data streams.
- Use a rectangular discretization in UV space and create two geometry instances: non-indexed and indexed.

Notes

- In most cases, the user will not need to access vertex data directly from the code. The purpose of this demo is to reinforce the structure concepts shown earlier.

D103B:Ex02 - Geometry

- **Conceptual design**
  - UV space: a natural surface-bound coordinate system that is used for discretizing the surface and generating texture coordinates (typically the same as UV).
  - UV space is also referred to as natural or curvilinear coordinate system. A reading in analytical geometry is useful for working in texturing and surface modeling.
  - Topologically, most surface discretizations in UV space are rectangular meshes. Here, the UV space is defined by the circumference and elevation angle of the torus.
- **The major point of the demo is not so much how the torus is discretized, but how we define the geometry and create its data structures in code.**

D103B:Ex02 - Geometry

- **Definition of geometry in the script. Note that we explicitly size the empty vertex sets, and tell Principia not to load them as it parses the script. The load step (data allocation and generation) will be done in the user code.**

```
##define (VTD_FVF_EX02) as <DC_VTXDEF3>
   #tag IsFVF      =  (           YES )
   #tag Format     = "POS:3F, NORM:3F, COLOR:COLOR, TEX:2F"
   #tag VtxMemory  =  (   GMEM_DEVICE )
   #tag IdxMemory  =  (   GMEM_DEVICE )
   #tag IndexFmt   =  (       INDEX16 )
   #tag IsDynamic  =  (             - )
   #tag IsWrtonly  =  (             - )

##define (G_EX02_NOINDEX) as <VERTEXSET_3A>
   #tag File       =  (         NONE )
   #tag Definer    =  ( VTD_FVF_EX02 )
   #tag Indexed    =  (           NO )
   #tag LoadNow    =  (           NO )
   #tag Geometry   =  (    G_TRILIST )
   #tag NbVertex   =  (            - )
   #tag NbFrames   =  (            - )

##define (G_EX02_INDEXED) as <VERTEXSET_3A>
   #tag File       =  (         NONE )
   #tag Definer    =  ( VTD_FVF_EX02 )
   #tag Indexed    =  (          YES )
   #tag LoadNow    =  (           NO )
   #tag Geometry   =  (    G_TRILIST )
   #tag NbVertex   =  (            - )
   #tag NbFrames   =  (            - )
```

- **There are two geometry instances: indexed and not.**

D103B:Ex02 - Geometry

- **User code: The generation of geometry with rectangular mesh topology is genericized and encapsulated:**
  - The mesh is parametrized in terms of normalized U and V coordinates, covering the interval [0,1]. Here, U is the circumferential azimuth.
  - The geometry terms of the U and V parameters is contained in a function that is passed as argument to the geometry generation routine.
  - To change the geometry, just change this function, technically known as a geometry kernel. The generation function can be left alone, or can be tweaked by the user to add extra features such as coordinate sets...etc.

```
void Pgv_Ex02_Init (void* VArg)
BGN_ACTOR_CODE{

   /* Auxiliary local variables */
   DECLARE_LOCAL  static   GX_VSet3A*       VSet_Noindex    = NULL;
   DECLARE_LOCAL  static   GX_VSet3A*       VSet_Indexed    = NULL;

   /* Predetermined size for torus discretization */
   DECLARE_LOCAL  const    int              NU      = 48;
   DECLARE_LOCAL  const    int              NV      = 48;

   /* Connect the vertex set to the script object */
   VSet_Noindex = (GX_VSet3A*) Principia->GetReferenceTo("G_EX02_NOINDEX");
   VSet_Indexed = (GX_VSet3A*) Principia->GetReferenceTo("G_EX02_INDEXED");

   /* Generate the vertex set geometry using a simple mesh distorted according to the kernel */
   Usr_GenerateTriangleVSet_FromKernel (VSet_Noindex, NU, NV, Usr_Geom_GreenCopperTorus, false);
   Usr_GenerateIndexedVSet_FromKernel  (VSet_Indexed, NU, NV, Usr_Geom_GreenCopperTorus, false);

END_ACTOR_CODE}
```

D103B:Ex02 - Geometry

Generation function, non-indexed geometry:

- Vertex stream is sized * GX_VSet3A is ::Loaded() to allocate buffers * Frame boundaries are defined * Vertex set is open for DMA access * Working vertex point linear array is generated by evaluating the kernel across UV ...

```
void Usr_GenerateTriangleVSet_FromKernel
(GX_VSet3A* VSet, int NU, int NV, void(*Geom3DKrn)(flt, flt, flt&, flt&, flt&, DWORD&, DWORD&, flt&, flt&), bool InvertN)
BGN_ACTOR_CODE{

   /* Auxiliary local variables */
   DECLARE_LOCAL  static   flt             SurfU,SurfV     = ZERO;
   DECLARE_LOCAL  static   int             ku,kv, kf,L, Fo = 0x00;

   /* Auxiliary local variables */
   DECLARE_LOCAL  static   flt*            x       = NULL;
   . . .

   /* Initialize working arrays for vertex data */
   SAFE_ALLOC(x ,  (NU+1)*(NV+1), flt);
   . . .

   /* Size and load the vertex set to create the GPU data buffers. Note that NU and NV count */
   /* subdivisions along the shape UV directions. Vertex counts are therefore NU+1 and NV+1. */
   VSet->NFrames    = 0x01;
   VSet->NVTot      = 6*NU*NV;
   VSet->Park();
   VSet->Load();

   /* Specify the frame boundaries and per-frame primitive count */
   VSet->FrameVO[0] = 0;
   VSet->FrameNP[0] = 2*NU*NV; // triangle count = 2xNUxNV
   VSet->FrameNV[0] = 6*NU*NV; // vertex count = 3xtriangle count

   /* Open vertex set buffer to data access */
   VSet->StartVrtexDma();

   /* Vertex set definition: vertices, texture coordinates and local colors */
   for (ku=0; ku<=NV; ku++) begin{
   for (kv=0; kv<=NU; kv++) begin{
       L      = kv*(NV+1) + ku;
       SurfU  = float(ku) / float(NV);
       SurfV  = float(kv) / float(NU);
       Geom3DKrn(SurfU,SurfV, x[L],y[L],z[L], cD[L],cS[L], u0[L], v0[L]);
   endfor}
   endfor}
```

D103B:Ex02 - Geometry

Generation function, non-indexed geometry:

- . . . Normals are calculated on the working mesh * The working linear array points are mapped to the vertex elements in the buffer. This is the procedure heart. Each UV patch has two triangles and six vertices * DMA access to the vertex set is closed to allow rendering * The rest is cleanup.

```
   /* Vertex set definition: vertex normals and enclosure correction */
   MeshNormals(Nx, Ny, Nz, x, y, z, 0, NV+1, 0, NU+1, NV+1, InvertN);

   /* Vertex set definition: fill vertex data buffer */
   Fo = VSet->FrameVO[0];
   for (ku=NV; ku>0; ku--) begin{
   for (kv=0; kv<NU; kv++) begin{
      L  = (kv)*(NV+1) + (ku);
      VSet->SetP (Fo+0, x [L], y [L], z [L]);
      VSet->SetN (Fo+0, Nx[L], Ny[L], Nz[L]);
      VSet->SetT0(Fo+0, u0[L], v0[L]);
      VSet->SetC (Fo+0, cD[L], cS[L]);
      . . .
      L  = (kv)*(NV+1) + (ku);
      VSet->SetP (Fo+5, x [L], y [L], z [L]);
      VSet->SetN (Fo+5, Nx[L], Ny[L], Nz[L]);
      VSet->SetT0(Fo+5, u0[L], v0[L]);
      VSet->SetC (Fo+5, cD[L], cS[L]);
      Fo = Fo + 6;
   endfor}
   endfor}

   /* Close vertex set buffer to data access */
   VSet->StopVrtexDma();

   /* Use this if setting a separate material for the torus */
   VSet->Material = NULL;

   /* Cleanup */
   SAFE_FREE (x);
   . . .

END_ACTOR_CODE}
```

D103B:Ex02 - Geometry

- **Geometry Kernel function defines the torus in UV space:**

```
void Usr_Geom_GreenCopperTorus(flt U, flt V, flt& x, flt& y, flt& z, DWORD& cD, DWORD& cS, flt &TexU, flt &TexV)
BGN_ACTOR_CODE{

   . . .
   DECLARE_LOCAL  const    flt             R1      = 0.70f;
   DECLARE_LOCAL  const    flt             R2      = 0.35f;

   /* Parametric representation: texture coordinates */
   TexU = U;
   TexV = V;

   /* Parametric representation: geometric shape */
   U     = 90.0f - 360.0f*U;
   V     = 360.0f*V;
   DR    = R2*DCOS(U);
   DZ    = R2*DSIN(U);
   PC    = DCOS(V);
   PS    = DSIN(V);
   RT    = R1+DR;
   x     = PC*RT;
   y     = PS*RT;
   z     = DZ;

   /* Parametric representation: surface colors */
   cD    = 0xFF22BB33;
   cS    = 0xFFFF1111;

END_ACTOR_CODE}
```

- **The kernel-parametric generator in the user code is a very simplified example of the powerful procedural generators provided by Principia. To use these, you do not need to write any code, just describe your kernel in script. Now you see the power of Principia!**

D103B:Ex02 - Geometry

- **Generation function, indexed geometry: The general flow is the same, but the sizing and mapping of vertex data reflects that the topology will be described in an index buffer.**

```
void Usr_GenerateIndexedVSet_FromKernel
(GX_VSet3A* VSet, int NU, int NV, void(*Geom3DKrn)(flt, flt, flt&, flt&, flt&, DWORD&, DWORD&, flt&, flt&), bool InvertN)
BGN_ACTOR_CODE{

   . . .
   /* Auxiliary local variables */
   DECLARE_LOCAL  static   flt*            x       = NULL;
   . . .

   /* Initialize working arrays for vertex data */
   SAFE_ALLOC(x ,  (NU+1)*(NV+1), flt);
   . . .

   /* Size and load the vertex set to create the GPU data buffers. Note that NU and NV count */
   /* subdivisions along the shape UV directions. Vertex counts are therefore NU+1 and NV+1. */
   VSet->Indexed    = true;          // In case we forgot to define it as indexed …
   VSet->NFrames    = 0x01;
   VSet->NVTot      = (NU+1)*(NV+1); // vertex count = NU+1 x NV+1 unique vertices
   VSet->NITot      = 6*NU*NV;       // vertices that need to be indexed is still 3xtriangle count
   VSet->Park();
   VSet->Load();

   /* Specify the frame boundaries and per-frame primitive count */
   VSet->FrameVO[0] = 0;            // first vertex at position 0
   VSet->FrameNP[0] = 2*NU*NV;      // triangle count
   VSet->FrameNV[0] = VSet->NVTot;  // vertex count
   VSet->FrameIO[0] = 0;            // first index at position 0
   VSet->FrameIN[0] = VSet->NITot;  // index count

   /* Open geometry buffers to data access */
   VSet->StartDmaMode();

   /* Vertex set definition: vertices, texture coordinates and local colors */
   for (ku=0; ku<=NV; ku++) begin{
   for (kv=0; kv<=NU; kv++) begin{
       L  = kv*(NV+1) + ku;
       SurfU  = float(ku) / float(NV);
       SurfV  = float(kv) / float(NU);
       Geom3DKrn(SurfU,SurfV, x[L],y[L],z[L], cD[L],cS[L], u0[L], v0[L]);
   endfor}
   endfor}
```

D103B:Ex02 - Geometry

- **Generation function, indexed geometry: The vertices are now uniquely defined and held in a linear array. Our index buffer holds the 3xTriangle count triangle-forming indexes.**

```
   /* Vertex set definition: vertex normals and enclosure correction */
   MeshNormals(Nx, Ny, Nz, x, y, z, 0, NV+1, 0, NU+1, NV+1, InvertN);

   /* Geometry definition: fill vertex data buffer */
   Fo = VSet->FrameVO[0];
   for (ku=0; ku<=NV; ku++) begin{
   for (kv=0; kv<=NU; kv++) begin{
      L  = (kv)*(NV+1) + (ku);
      VSet->SetP (Fo, x [L], y [L], z [L]);
      VSet->SetN (Fo, Nx[L], Ny[L], Nz[L]);
      VSet->SetT0(Fo, u0[L], v0[L]);
      VSet->SetC (Fo, cD[L], cS[L]);
      Fo++;
   endfor}
   endfor}

   /* Geometry definition: fill index data buffer */
   Fo = VSet->FrameIO[0];
   for (ku=0; ku<NV; ku++) begin{
   for (kv=0; kv<NU; kv++) begin{
      L  = (kv)*(NV+1) + (ku);
      VSet->SetIndex(Fo, L     ); Fo++;
      VSet->SetIndex(Fo, L+1   ); Fo++;
      VSet->SetIndex(Fo, L+NU+2); Fo++;
      VSet->SetIndex(Fo, L     ); Fo++;
      VSet->SetIndex(Fo, L+NU+2); Fo++;
      VSet->SetIndex(Fo, L+NU+1); Fo++;
   endfor}
   endfor}

   /* Close geometry buffers to data access */
   VSet->StopDmaMode();

   /* Use this if setting a separate material for the torus */
   VSet->Material = NULL;

   /* Cleanup */
   SAFE_FREE (x);
   . . .

END_ACTOR_CODE}
```

D103B:Ex02 - Geometry

Vertex data accessor functions.

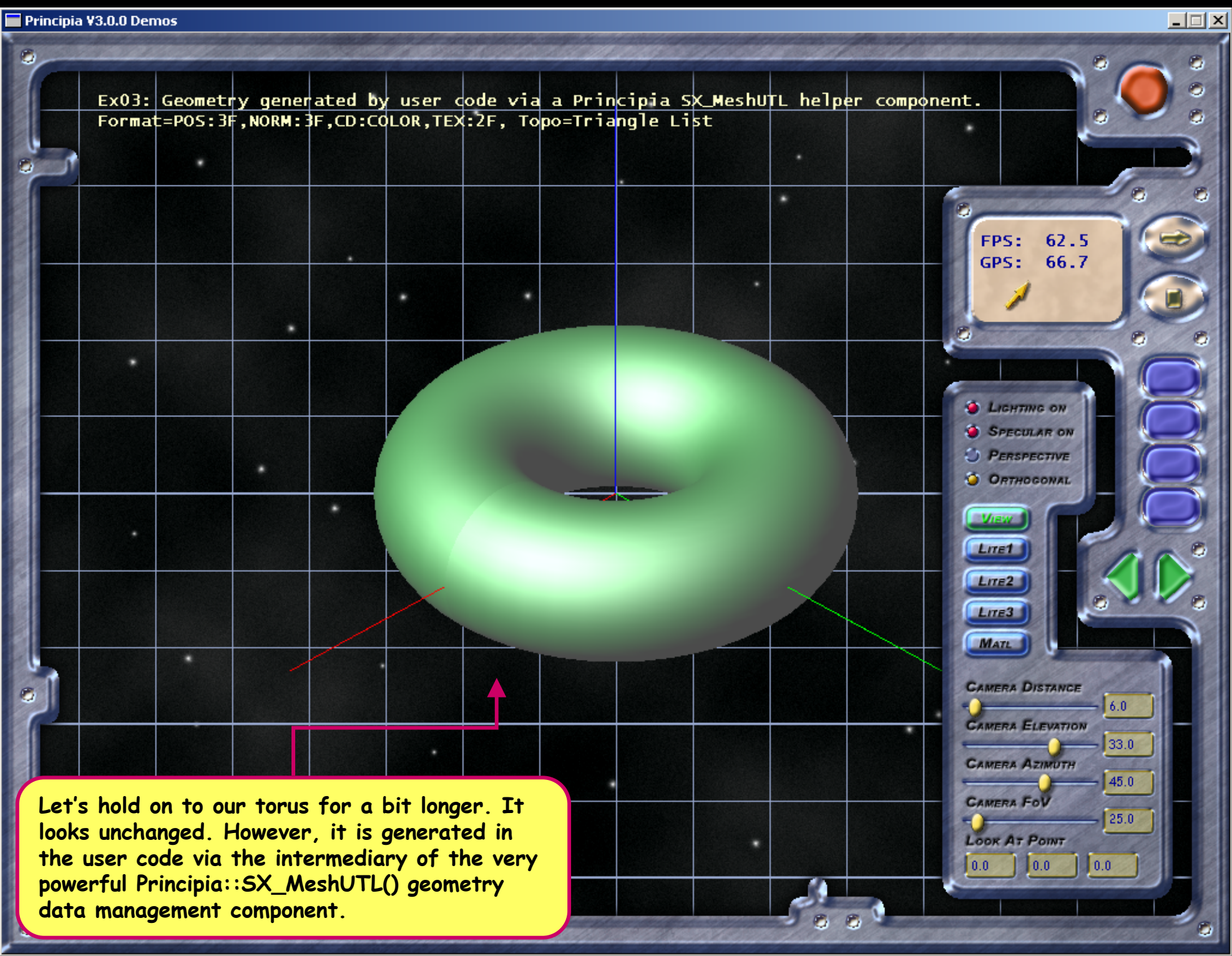
- Principia provides many functions to access vertex and index buffer data streams. To use these functions, the vertex set component must be opened for DMA access in the Api, and DMA-closed before rendering (using ::Start/StopDMAMode() methods).

Common vertex data access functions, block transfer:

- void Buffer (void* ExtData, int NbVertices);
- void Buffer (void* ExtData, int VtxIdxA, int VtxIdxB);
- void Export (void* ExtData, int NbVertices);
- void Export (void* ExtData, int VtxIdxA, int VtxIdxB);
- void Set/GetP (int NA, int NB, flt* x, flt* y, flt* z);
- void Set/GetCD (int NA, int NB, int kF, DWORD* Cd); ... etc.

Common vertex data access functions by element:

- flt* Get/SetVertex (int i);
- int Get/SetIndex (int i);
- void Get/Set (int i, flt& D, int Ofs);
- void Get/SetP (int i, flt& x, flt& y, flt& z);
- void Get/SetN (int i, flt& x, flt& y, flt& z);
- void Get/SetC (int i, DWORD& Cd, DWORD& Cs);
- void Get/SetT0/...T7 (int i, flt& u, flt& v);
- void Get/SetT (int cDex, int i, flt& u, flt& v, flt& w, flt& r);
- void Get/SetB (int i, flt& B0, flt& B1, flt& B2, DWORD& BW);
- void Get/SetX (int i, flt& S); ... etc.


Principia V3.0.0 Demos
Ex03: Geometry generated by user code via a Principia SX_MeshUTL helper component.
Format=POS:3F,NORM:3F,CD:COLOR,TEX:2F, Topo=Triangle List
FPS: 62.5
GPS: 66.7
Lighting on
Specular on
Perspective
Orthogonal
View
Lite1
Lite2
Lite3
Matl
Camera Distance
6.0
Camera Elevation
33.0
Camera Azimuth
45.0
Camera FoV
25.0
Look At Point
0.0
0.0
0.0
Let's hold on to our torus for a bit longer. It looks unchanged. However, it is generated in the user code via the intermediary of the very powerful Principia::SX_MeshUTL() geometry data management component.

D103B:Ex03 - Geometry

Objectives

- Show a more advanced modality for creating and managing geometry data from within the user code.

Requirements and Implementation

- Define a renderable Mesh3A in the script and leave its geometry field blank
- In the user code, create the geometry using the Principia SX_MeshUTL (unstructured mesh) internal component. Use whatever format you want.
- Create the vertex set from the SX_MeshUTL, and add it to the renderable Mesh3A from the script. Render!

Notes

- This is a much better, faster and more flexible way to create and manage data from within the user code.

D103B:Ex03 - Geometry

- **SX_MeshUTL()**
  - General purpose unstructured mesh internal Principia component. Workhorse for code geometry operations.
  - Mesh defined as a series of 3D data points, with triangle face connectivity list (like an indexed VSet3A). This is the simplest standard mesh definition.
  - Isolates the user from having to worry about managing format data fields and low-level tasks.
  - Features tons of topological, geometry and data access operations.
  - Provides easy animation frame access.
- **SX_MeshUTL() can be easily converted into different GX_VSet() components and vice-versa.**

D103B:Ex03 - Geometry

- **This is our script. We do not even need to define the geometry object therein, since it will be created internally.**

```
##define (M_EX03) as <MATERIAL_3A>
   #tag RS         =  (               RS_FILLMODE ,        RA_SOLID )
   #tag RS         =  (               RS_CULLMODE ,      RA_CULLCCW )
   #tag TS         =  (                 TS_COLOROP ,   TO_SELECTARG2 ,   0 )
   #tag TS         =  (              TS_COLORARG1 ,      TA_TEXTURE ,   0 )
   #tag TS         =  (              TS_COLORARG2 ,      TA_DIFFUSE ,   0 )
   #tag TS         =  (                TS_ALPHAOP ,   TO_SELECTARG2 ,   0 )
   #tag TS         =  (              TS_ALPHAARG1 ,      TA_TEXTURE ,   0 )
   #tag TS         =  (              TS_ALPHAARG2 ,      TA_DIFFUSE ,   0 )

##define (R_EX03) as <MESH_3A>
   #tag Component  =  (           0 ,      M_EX03 ,     NONE )

##define (O_EX03) as <GENOBJECT_3A>
   #tag Construct  =  ( "Decl(M);Mul(0.5,0.5,0.5);" ,  R_EX03 , NONE )

##define (WRLD_EX03) as <WORLD_3A>
   #tag Layer_Decl =  (     L_ITEMS ,   "ITEMS" , - , - , 16 )
   #tag Layer_Item =  (     L_ITEMS ,                O_EX03 )
   #tag Layer_Item =  (     L_ITEMS ,             O_AXISYS )
```

D103B:Ex03 - Geometry

- **User code: The generation process is encapsulated in a function that uses the same kernel as Ex02. Note that we need to add the generated geometry to the <R_EX03> renderable mesh defined in script.**

```
void Pgv_Ex03_Init (void* VArg)
BGN_ACTOR_CODE{

   /* Auxiliary local variables */
   DECLARE_LOCAL  static   GX_Mesh3A*       Mesh           = NULL;
   DECLARE_LOCAL  static   GX_VSet3A*       VSet           = NULL;

   /* Predetermined size for torus discretization */
   DECLARE_LOCAL  const    int              NU      = 48;
   DECLARE_LOCAL  const    int              NV      = 48;

   /* Generate the vertex set geometry using the Principia mesh helper component */
   Usr_GenerateTriangleVSet_FromKernel_PrincipiaMeshHelper (VSet, NU, NV, Usr_Geom_GreenCopperTorus, false);

   /* Connect the vertex set to the script objects so that it can render */
   Mesh = (GX_Mesh3A*) Principia->GetReferenceTo("R_EX03");
   Mesh->Element[0] = VSet;

END_ACTOR_CODE}
```

D103B:Ex03 - Geometry

- **The generation function is much simpler and cleaner too.**

```
void Usr_GenerateTriangleVSet_FromKernel_PrincipiaMeshHelper
(GX_VSet3A* &VSet, int NU, int NV, void(*Geom3DKrn)(flt, flt, flt&, flt&, flt&, DWORD&, DWORD&, flt&, flt&), bool InvertN)
BGN_ACTOR_CODE{

   /* Auxiliary local variables */
   DECLARE_LOCAL  static   flt             SurfU,SurfV          = ZERO;
   DECLARE_LOCAL  static   int             ku,kv,L,NbVert,NbTrig = 0x00;
   DECLARE_LOCAL  DX_VtxDef3*              WrkVDef              = NULL;
   DECLARE_LOCAL  SX_MeshUTL*              WrkMesh              = NULL;

   /* Initialize definer and mesh components */
   NbVert  = (NU+1)*(NV+1);
   NbTrig  = 2*NU*NV;
   WrkVDef = new DX_VtxDef3("POS:3F, NORM:3F, COLOR:COLOR, TEX:2F");
   WrkMesh = new SX_MeshUTL(NbVert, NbTrig, WrkVDef);

   /* WrkMesh definition: point positions, texture coordinates and local colors */
   for (ku=0; ku<=NV; ku++) begin{
   for (kv=0; kv<=NU; kv++) begin{
       L      = kv*(NV+1) + ku;
       SurfU  = float(ku) / float(NV);
       SurfV  = float(kv) / float(NU);
       Geom3DKrn(SurfU,SurfV, WrkMesh->x[L],WrkMesh->y[L],WrkMesh->z[L], WrkMesh->CD[L], . . . WrkMesh->v0[L]);
   endfor}
   endfor}

   /* WrkMesh definition: face topology description via indexing */
   int Fo = 0x00;
   for (ku=0; ku<NV; ku++) begin{
   for (kv=0; kv<NU; kv++) begin{
      L  = (kv)*(NV+1) + (ku);
      WrkMesh->F[Fo] = L     ; Fo++;
      WrkMesh->F[Fo] = L+NU+2; Fo++;
      WrkMesh->F[Fo] = L+1   ; Fo++;
      WrkMesh->F[Fo] = L     ; Fo++;
      WrkMesh->F[Fo] = L+NU+1; Fo++;
      WrkMesh->F[Fo] = L+NU+2; Fo++;
   endfor}
   endfor}

   /* Calculate the mesh normals */
   WrkMesh->SurfaceNormals(InvertN, true /*smooth seams*/);

   /* Generation of the vertex set component and cleanup */
   VSet = new GX_VSet3A(WrkMesh, WrkVDef, true /*indexed*/);
   SAFE_DELETE (WrkMesh);
   . . .
END_ACTOR_CODE}
```

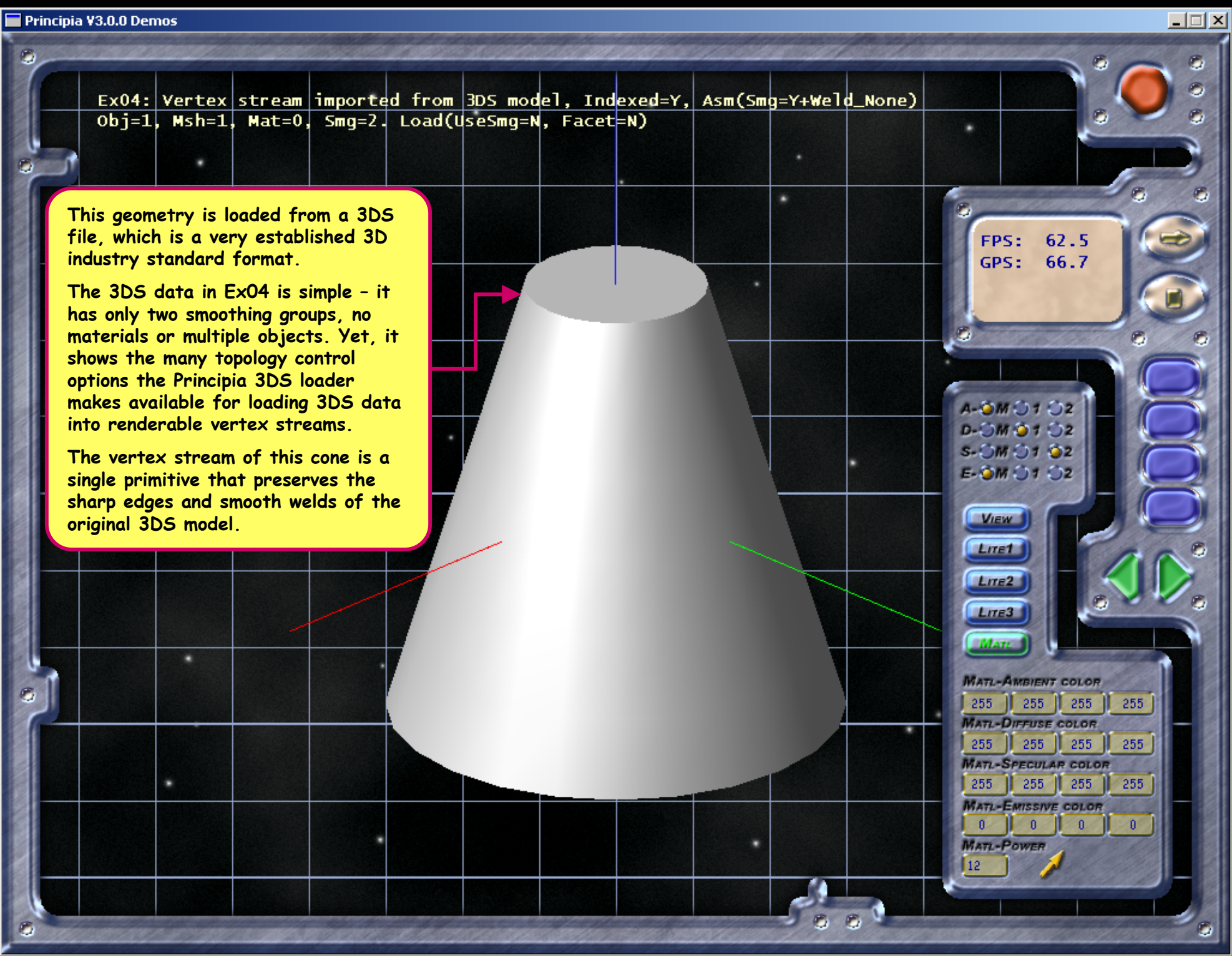
D103B:Ex03 - Geometry

Some common vertex stream creation methods:

- CONS GX_VSet3A (SX_Stream* CfgStream);
- CONS GX_VSet3A (char* FName, DX_VtxDef3* iDef);
- CONS GX_VSet3A (GX_VSet3A* ProtoObj);
- CONS GX_VSet3A (DX_VtxDef3* iDef);
- CONS GX_VSet3A (char* Fmt, char* Geom, int NPF, int NVF, int NF);
- CONS GX_VSet3A (SX_MeshUTL* SrcMesh, DX_VtxDef3* iDef, bool AsIndexed);

Some common SX_MeshUTL() methods:

- CONS SX_MeshUTL (int NbVtces, int NbFaces, DX_VtxDef3* VtxDef);
- CONS SX_MeshUTL (SX_MeshRUV* SrcMesh, DWORD* Mask, DX_VtxDef3* VtxDef);
- CONS SX_MeshUTL (GX_VSet3A *VtxSet, int FrameIdx);
- CONS SX_MeshUTL (SX_MeshUTL* SrcMesh);
- void SurfaceNormals/Tangent (bool Invert, bool SmoothSeam);
- void FaceNormals/Tangent (bool Invert);
- void Condense (bool PreserveSeams);
- int Map_CommonVertices (int FaceA, int FaceB, int &ComVA, int& ComVB, int &ComVC);
- void Map_AdjacentFaces ();
- void MaskedExtract (SX_MeshRUV* SrcMesh, DWORD* Mask);
- void DisplaceMap (SX_FunctionS* fU, SX_FunctionS* fV, SX_FunctionS* fN);
- void Attach (SX_MeshUTL* Adder, bool MergeB, flt MergeD);
- int DistanceToMesh/Edge (flt X, flt Y, flt Z, flt& D);
- void SplitUVSeams (flt CritU, flt CritV);
- void AddUniqueEdge (int v0, int v1);
- void WrteVSetFrame (GX_VSet3A* VtxSet, int FrameIdx);
- void WrteVSetStream (GX_VSet3A* VtxSet, flt* Data, int OffsetAddress, int FrameIdx);
- void WrteVSetNormals (GX_VSet3A* VtxSet);


Principia V3.0.0 Demos
Ex04: Vertex stream imported from 3DS model, Indexed=Y, Asm(Smg=Y+Weld_None)
Obj=1, Msh=1, Mat=0, Smg=2. Load(UseSmg=N, Facet=N)
This geometry is loaded from a 3DS file, which is a very established 3D industry standard format.
The 3DS data in Ex04 is simple – it has only two smoothing groups, no materials or multiple objects. Yet, it shows the many topology control options the Principia 3DS loader makes available for loading 3DS data into renderable vertex streams.
The vertex stream of this cone is a single primitive that preserves the sharp edges and smooth welds of the original 3DS model.
FPS: 62.5
GPS: 66.7
A- M 1 2
D- M 1 2
S- M 1 2
E- M 1 2
View
Lite1
Lite2
Lite3
Matl
Matl-Ambient color
255 255 255 255
Matl-Diffuse color
255 255 255 255
Matl-Specular color
255 255 255 255
Matl-Emissive color
0 0 0 0
Matl-Power
12

D103B:Ex04 - Geometry

Objectives

- Show how to load 3DS data in vertex streams and how to control their basic topology and structure.
- Show the recursive structure of GX_VSet3A and how it fits in the Principia *3A* rendering framework.

Key Requirements

- Load and draw a simple 3DS object with two smoothing groups and no materials.
- Explore how the topology of loaded vertex data is subtly (or not so subtly) controlled by the user.

Notes

- Principia features the best 3DS loader available! It can also load geometry data from many other popular 3D formats using its array of configurable procedures.

D103B – 3DS Max Files

- **3DS files are complex constructs with multiple:**
  - Objects
  - Meshes per object
  - Composite materials per mesh
  - Smoothing groups (fragments of smooth surface)
- **Loading 3DS files into GX_VSet3A objects**
  - Done by the intermediary of a Principia FX_Import3DS procedure, which can be specified in script or else internally generated.
  - Depending on the procedure configuration, multiple smoothing groups, materials, meshes... from the 3DS file etc, will result in a composite GX_VSet3A object, with an embedded subset of additional GX_VSet3A components (with their own materials if applicable).
  - GX_VSet3A loaded from 3DS files can be saved into VSF format. If present, each embedded component will be saved into a separate VSF file with a distinctive name.

D103B – 3DS Max Files

FX_Import3DS loading procedure:

- Stage 1: Load each “distinct geometry fragment” from the 3DS file into an individual MeshUTL object. Such fragments are characterized by separate smoothing groups, material, mesh and object.
- Stage 2: Optional assembly of the MeshUTL fragments from stage 1 into bigger fragments combining objects, materials ...etc based on user “welding” parameters for geometry, normals ...etc.
- Stage 3: Optional transfer of the resulting MeshUTL data into a renderable GX_VSet3A component with embedded GX_VSet3A/material structures if needed.

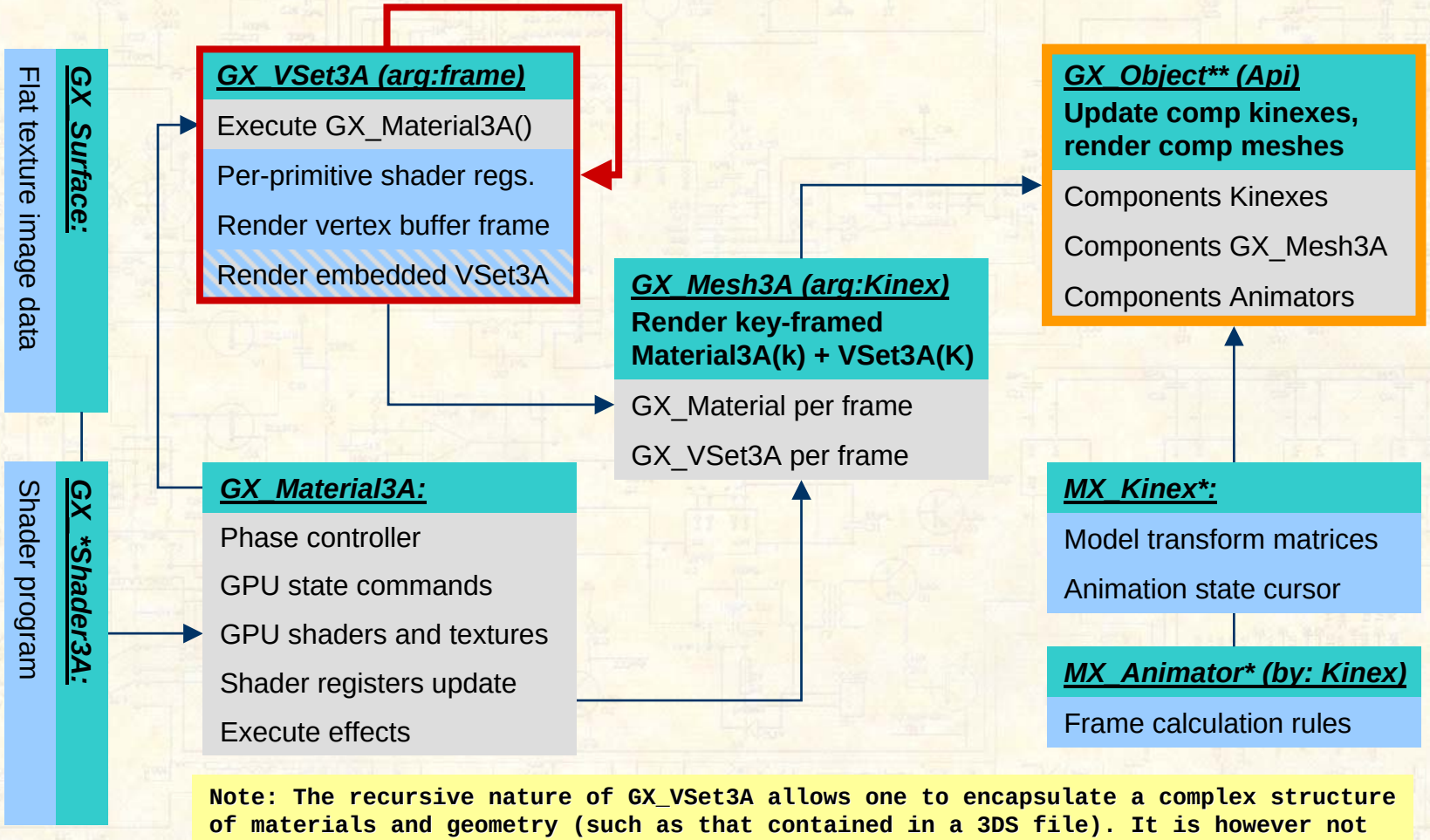
The power of the FX_Import3DS procedure:

- It gives the user very broad control over the topological mapping of the complex 3DS data into simple working Principia structures.
- Many Principia components use the FX_Import3DS procedure to access its MeshUTL data in its full detail. So can you!

D103B – 3DS Max Files

- **Why fine-tune how all the distinct 3DS geometry fragments are loaded into vertex data?**
  - Fragment structure and performance are closely linked.
  - Performance hit#1: Today's GPUs hate material changes.
  - Performance hit#2: Today's GPUs render a long primitive faster than many small primitives (even with the same triangle count).
  - Performance hit#3: Index and reduce redundant vertex data.
- **Basic production organization suggestions:**
  - Use indexed output unless dictated by artistic imperatives.
  - Assemble smoothing group fragments into single vertex primitives with seam preservation (or dispense with groups altogether).
  - Assemble different material fragments into single vertex primitives and bake the materials onto textures.
  - Make material changes as little as possible by grouping same-material geometry renders together.

D103B – Embedded GX_VSet3A



Note: The recursive nature of GX_VSet3A allows one to encapsulate a complex structure of materials and geometry (such as that contained in a 3DS file). It is however not an efficient way to render in high-performance applications, where is makes more sense to group geometry by material. The Principia *3A* rendering framework provides enough flexibility to configure the rendering sequence either way.

D103B – 3DS Loader Procedure

- **The Principia 3DSLoader procedure:**
  - Powerful tool for extracting 3D model content from 3DS files, storing it in manageable and accessible data chunks, and organizing it for 3D engine use.
  - Featuring multiple extraction modes depending on the content usage. Chapter 3 covers only the geometry extraction mode.
- **3DS extraction modes and control tags:**
  - Geometry (Exec_Geom tag, default=on)
  - Animation keys (Exec_Anim tag, default=off)
  - Bones hierarchy (Exec_Bone tag, default=off)

D103B – 3DS Loader Procedure

General control and output arguments:

- <ExVertexSet>: Output vertex stream. Will be automatically provided if the procedure is used as a loader in a GX_VSet3A definition (as here). Carries output vertex format and indexation selection.
- <Transform>: A kinex encoding a transform to be applied to all vertices loaded from the 3DS file. Note that the raw XYZ data exported in the 3DS file is that of the mesh geometry at Frame0 relative to the world coordinate system with the pivot system in its default MAX location. *If the pivot axis is manipulated in MAX, the affected geometry will not load properly.*
- <NeutralPose>: An advanced argument, used when importing complex skinned and/or animated characters. Covered in Part III, Chapters 6 and 12.
- <FragDefiner>: A definer for the loaded mesh fragments structure (instead of the default "POS:3F, NORM:3F, TEX:2F"). Use when you want to create vertex stream data placeholders for things such as skinning data or occlusion factors. What happens if conflict b/w FragDefiner and Extant VertexDefiner?
- If the vertex output is NULL, the 3DS loader procedure will still load all MeshUTL fragments and assemble them as per stage 1 and 2 settings. In this mode, the 3DS loader procedure is used by a host of other Principia geometry procedures (e.g. character generators) that load raw MAX data as input.

D103B – 3DS Loader Procedure

- **Stage 1 arguments, primary MeshUTL fragments loader**
  - FilterInObjName: If supplied, only fragments from the named objects from the 3DS file will be loaded. Multiple include names can be specified with standard Principia string match flags, wildcards and tolerances.
  - FilterExObjName: If supplied, will not load any fragments from the named objects from the 3DS file. Use to exclude irrelevant geometry.
  - FilterInMtlName: If supplied, only fragments having the named material will be loaded.
  - FilterExMtlName: Will not load any fragments having the named material. Note that these filters will work if the <UseMtl> tag is turned off. Use stage 3 assembly to combine the loaded fragments after material filtering.
  - Object and material filters can be combined – e.g. load only the “eyelash” material geometry fragments from the “head” object. Object filters are applied before material filters. This matters for exclude filters only. If an object containing a filter-desired material is excluded, that little piece of material will not be loaded.

D103B – 3DS Loader Procedure

- **Stage 1 arguments (continued…)**
  - Ldo_MeshFacetize: If set, all faces of the input geometry will have distinct sharp edges (regardless of vertex stream indexation and topology)
  - Ldo_MeshUseSmg: Set to false to ignore smoothing group structure when defining fragments. Use when you want speed over precision.
  - Ldo_MeshUseMtl: Set to false to ignore 3DS material groups when defining fragments. Use when the material subdivisions do not carry topological information per se, and you plan to condense the mesh anyway (e.g. in Principia character rigging procedures)
  - Ldo_FragNormals: Standard normal flags processing per fragment.
    - Invert normals. 3DS files do not carry normal data. This is calculated by for each MeshUTL fragment. Set to INVALID to skip per-fragment normal generation.
    - Normals averaging across seams: Each MeshUTL fragment can have internal seams with vertex position and/or UV duplication. This will average the normals across the seams (but preserve the seams)
  - Ldo_FragMeshWeld: Provides standard code and tolerance for optional welding of fragment MeshUTL internal seams and redundant vertices.
  - Typical codes are WELD_NONE (default), WELD_POS (condense all vertices with coincident positions even if their UV are different thereby closing all seams), and WELD_POSUV (condense vertices with same position and UV)

D103B – 3DS Loader Procedure

- **Stage 2 arguments, MeshUTL fragment assembly**
  - Asm_XSmg: Flag and seam weld code/tolerance for combining fragments across smoothing groups (material groups and objects are preserved).
  - Asm_XMat: Flag and seam weld code/tolerance for combining fragments across materials (smoothing groups and objects are preserved).
  - Asm_XObj: Flag and seam weld code/tolerance for combining fragments across meshes and objects (smoothing groups and materials are preserved)
  - Asm_Normals: Standard normals processing code for the MeshUTL resulting from the assembly. Mainly used to smooth post-assembly seams.
  - Setting all flags on welds the 3DS data into a single MeshUTL and a single vertex stream. With pre-baked materials, this is usually the best way.
- **Stage 2 arguments, MeshUTL fragment postprocessing**
  - Mod_UV: Sometimes, MAX will invert texture V-coordinates when exporting to 3DS format (welcome to MAX, this is not the only bug it has...). Whether it is to correct that, or to procedurally modify imported texcoords, you can specify a mesh_name/name_match_code/U/V_modfcn(u,v,x,y,z) procedural functions that will modify the texcoords for select meshes (or "*" for all).

D103B – 3DS Loader Procedure

- **Stage 3 arguments, fragments to (nested) vertex stream**
  - VgnNamePrefix: If the output stream has no name, it will be used to generate the name of each GX_Vset component and MeshUTL fragment as follows: VgnNamePrefix_Obj3DS[name]_Mesh3DS[#]_Material3DS[name]_Smg[#]
  - Vgn_StripMtl: Do not map any 3DS materials from the 3DS file onto the vertex stream. Used to import geometry only.
  - Vgn_PureObjNames: When generating vertex stream names, use only the name of the 3DS object from which the stream fragment came. Useful when the vertex output will be passed to other procedures that refer to fragments by their 3DS object name (we do not want material...etc. clutter in the name).

D103B:Ex04 - Geometry

- **Loading a 3DS file: Configure a loader procedure and pass it to the vertex stream ##define, along with the 3DS file you wish to load.**

```
##define (PROC_3DSLD_EX04) as <PROCDAT_IMPORT3DS>
   #tag Transform         = "Decl(M);Mul(1.00,1.00,1.00);"
   #tag Ldo_MeshFacetize  = (   NO )
   #tag Ldo_MeshUseSmg    = (  YES )
   #tag Ldo_FragNormals   = (   NO ,         YES )
   #tag Ldo_FragMeshWeld  = (         WELD_POSUV , 0.0 )
   #tag Asm_XSmg          = (  YES , WELD_NONE  , 0.0 )
   #tag Asm_XMat          = (   NO , WELD_NONE  , 0.0 )
   #tag Asm_XObj          = (   NO , WELD_NONE  , 0.0 )

##define (G_EX04) as <VERTEXSET_3A>
   #tag File      =  "Files_Media\Mesh_Cone.3ds"
   #tag Definer   =  (    VTD_FVF_FILE )
   #tag LoadProc  =  ( PROC_3DSLD_EX04 )
   #tag Indexed   =  (             YES )
   #tag LoadNow   =  (               1 )
```

- **If you create a vertex stream from a 3DS file without providing a procedure, Principia will create a default-setting loader for you (but the results may not be what you want).**

D103B:Ex04 - Geometry

Best-practice 3DS loader procedure settings:

- Aim to reduce the vertex stream primitive count loaded and material changes. To this end, examine the core dump file to see all the chunks that are in the 3DS file being loaded.

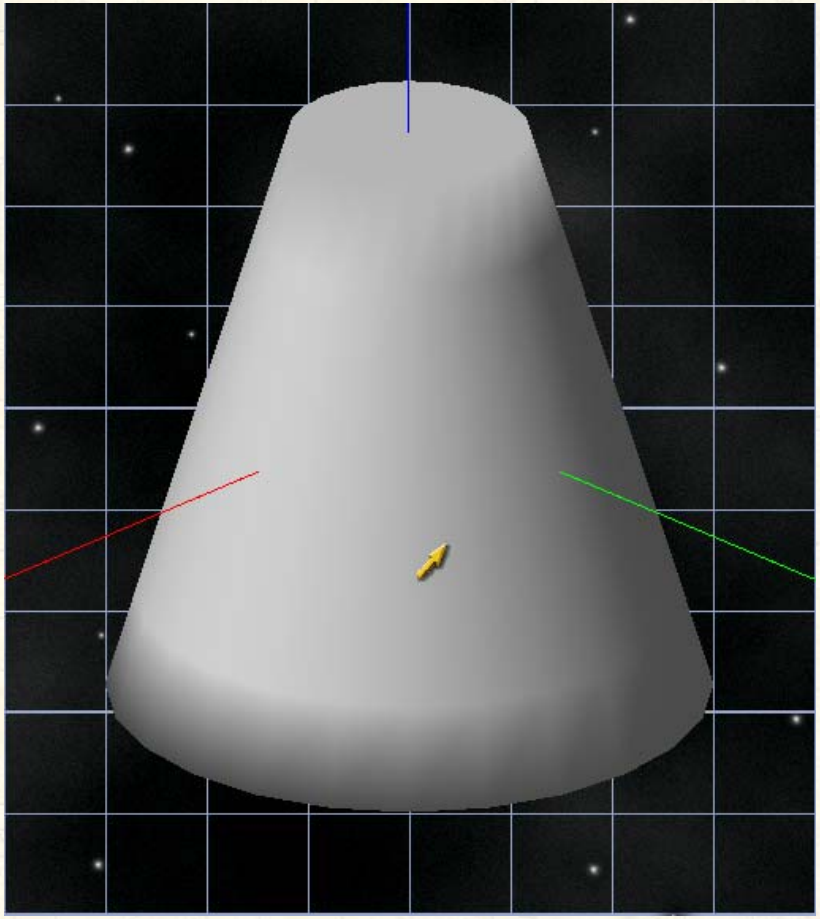
```
CH:0x4D4D: Ver(3)
CH:0xDATA: Data(10193)
CH:0x3D3E: Len(10)
CH:0x100: Len(10)

CH:0x4000: Obj(Cone01)
CH:0x4100: Len(10154)
CH:0x4110: NbV(244)
CH:0x4140: NbT(244)
CH:0x4160: Mtx(     0.0000,     0.0000,11.4f)
CH:0x4120: NbF(432)
CH:0x4150: Smg(1)
CH:0xB000: Len(220)
```

- Here, there are only two smoothing groups. We set the “Use Smoothing Groups” flag to yes. Otherwise, the top and bottom of the cylinder will not have sharp seams.
- We also set the “Assemble Smoothing Groups” flag to yes, but turn off welding. This preserves the sharp seam, but welds the fragment meshes into a single vertex primitive.

D103B:Ex04 - Geometry

- If we use a “WELD_POS” code when assembling across smoothing groups, the resulting primitive loses its sharp edges.

```
##define (PROC_3DSLD_EX04) as <PROCDAT_IMPORT3DS>

   #tag Transform         = "Decl(M);Mul(1.00,1.00,1.00);"
   #tag Ldo_MeshFacetize  = (   NO )
   #tag Ldo_MeshUseSmg    = (  YES )
   #tag Ldo_FragNormals   = (   NO ,         NO )
   #tag Ldo_FragMeshWeld  = (          WELD_POS , 0.0 )
   #tag Asm_XSmg          = (  YES ,  WELD_POS , 0.0 )
   #tag Asm_XMat          = (   NO , WELD_NONE , 0.0 )
   #tag Asm_XObj          = (   NO , WELD_NONE , 0.0 )
```



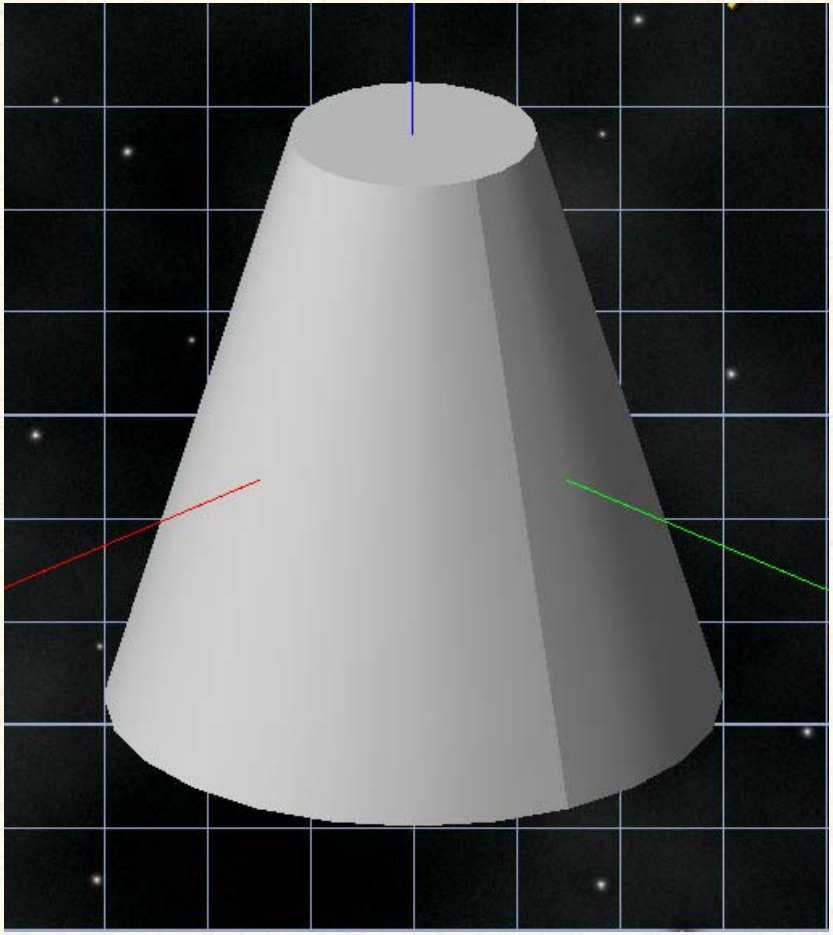
- Note the importance of managing smoothing groups when refining the renders of your models. Smoothing groups are topologically separate regions delimited by sharp edges.

D103B:Ex04 - Geometry

- If we do not specify an internal fragment condensation (before assembly), the seam that is within the first smoothing group (cone side) will not be closed.

```
##define (PROC_3DSLD_EX04) as <PROCDAT_IMPORT3DS>

   #tag Transform         = "Decl(M);Mul(1.00,1.00,1.00);"
   #tag Ldo_MeshFacetize  = (   NO )
   #tag Ldo_MeshUseSmg    = (  YES )
   #tag Ldo_FragNormals   = (   NO ,         NO )
   #tag Ldo_FragMeshWeld  = (        WELD_NONE , 0.0 )
   #tag Asm_XSmg          = (  YES , WELD_NONE , 0.0 )
   #tag Asm_XMat          = (   NO , WELD_NONE , 0.0 )
   #tag Asm_XObj          = (   NO , WELD_NONE , 0.0 )
```

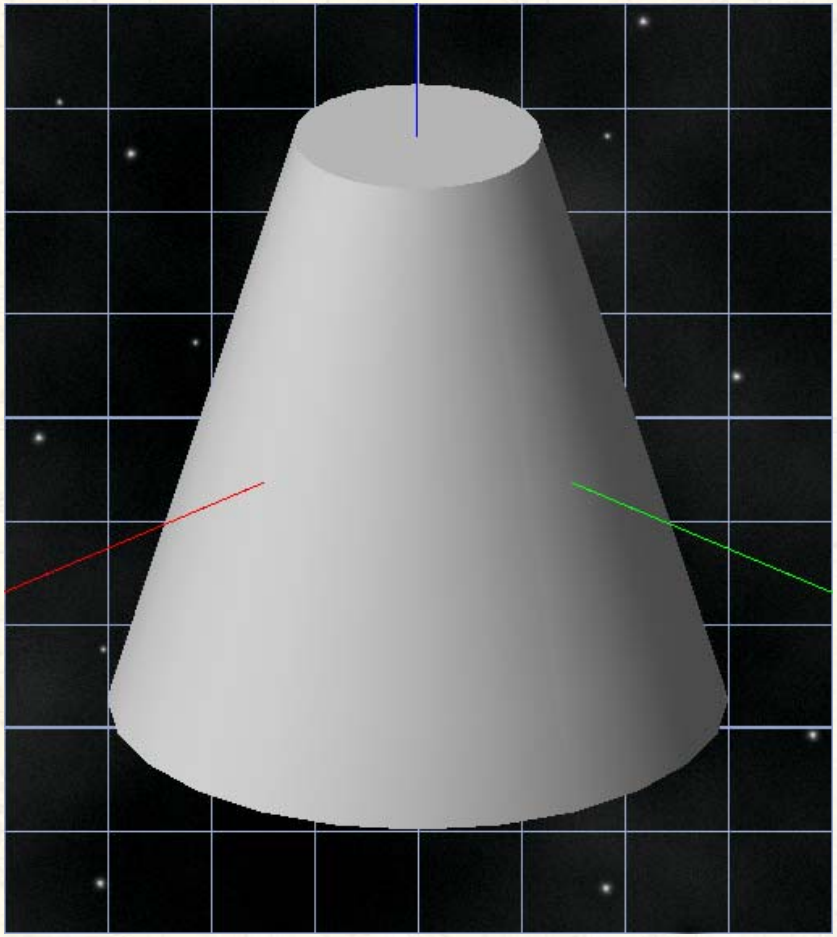
- Internal fragment condensation codes control seams within Stage1 fragments. Assembly codes control seams between fragments being attached together in Stage3.



D103B:Ex04 - Geometry

- We may want to preserve internal fragment seams (e.g. for texturing purposes) but have smooth normal across them. To achieve this, set the fragment seam normal averaging flag.

```
##define (PROC_3DSLD_EX04) as <PROCDAT_IMPORT3DS>

   #tag Transform         = "Decl(M);Mul(1.00,1.00,1.00);"
   #tag Ldo_MeshFacetize  = (   NO )
   #tag Ldo_MeshUseSmg    = (  YES )
   #tag Ldo_FragNormals   = (   NO ,       YES )
   #tag Ldo_FragMeshWeld  = (       WELD_NONE , 0.0 )
   #tag Asm_XSmg          = (  YES , WELD_NONE , 0.0 )
   #tag Asm_XMat          = (   NO , WELD_NONE , 0.0 )
   #tag Asm_XObj          = (   NO , WELD_NONE , 0.0 )
```



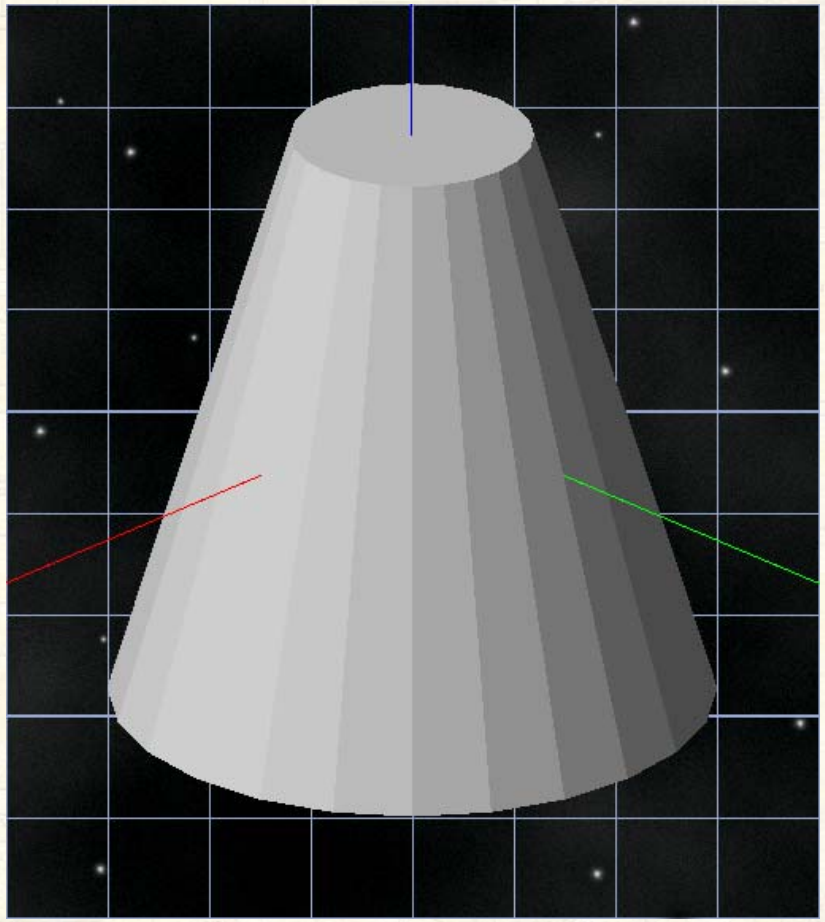
- A seam is basically a region where more than two vertices have the same position. Many Principia procedures take seam control flags similar to that in Import3DS.

D103B:Ex04 - Geometry

- **When the <Facetize> tag is set, all MeshUTL fragments have faces with distinct vertices (unless you have set intra-fragment normal seam averaging on).**

```
##define (PROC_3DSLD_EX04) as <PROCDAT_IMPORT3DS>

   #tag Transform         = "Decl(M);Mul(1.00,1.00,1.00);"
   #tag Ldo_MeshFacetize  = (  YES )
   #tag Ldo_MeshUseSmg    = (  YES )
   #tag Ldo_FragNormals   = (   NO ,         NO )
   #tag Ldo_FragMeshWeld  = (        WELD_NONE , 0.0 )
   #tag Asm_XSmg          = (  YES , WELD_NONE , 0.0 )
   #tag Asm_XMat          = (   NO , WELD_NONE , 0.0 )
   #tag Asm_XObj          = (   NO , WELD_NONE , 0.0 )
```

- **The output vertex stream may be indexed (as here) or not. The topology will still correctly load from the 3DS file based on your procedure settings.**

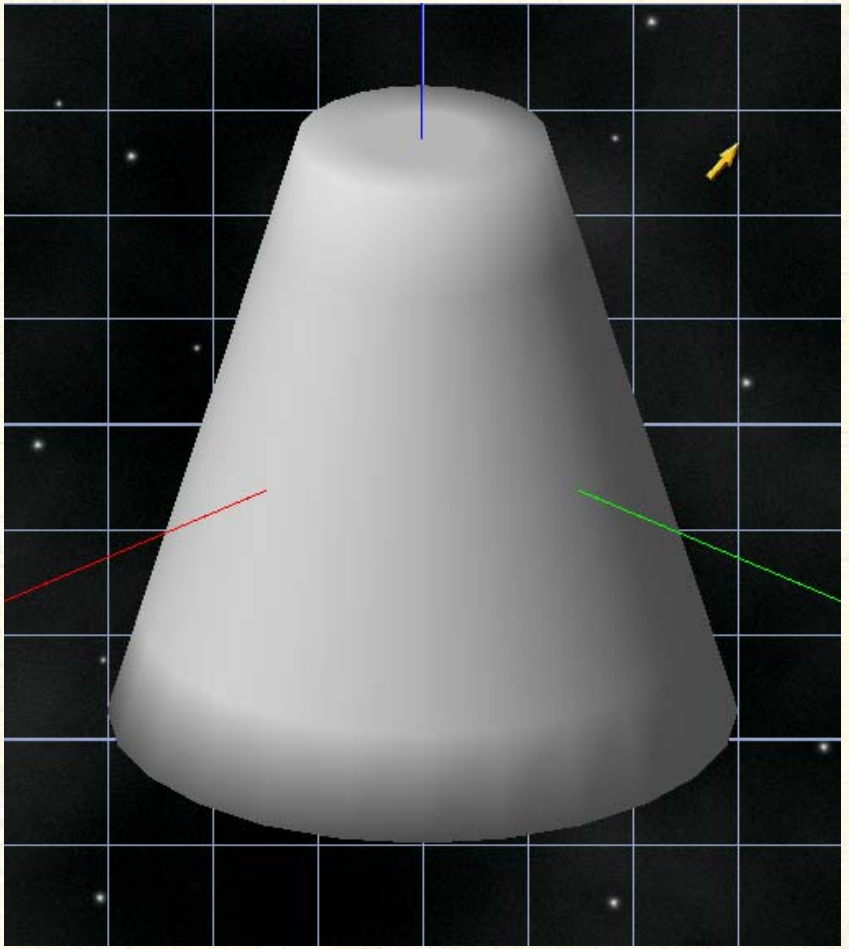


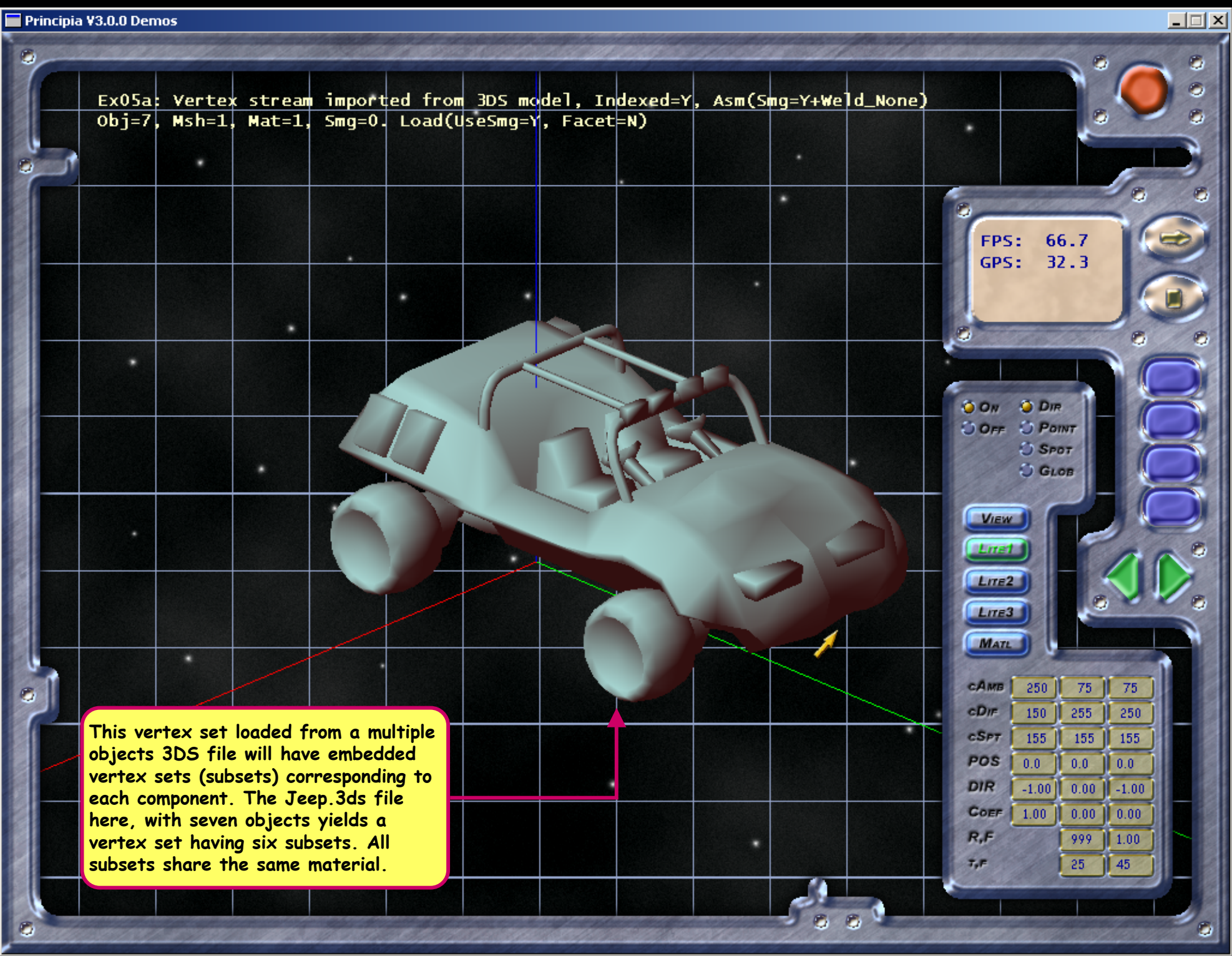
D103B:Ex04 - Geometry

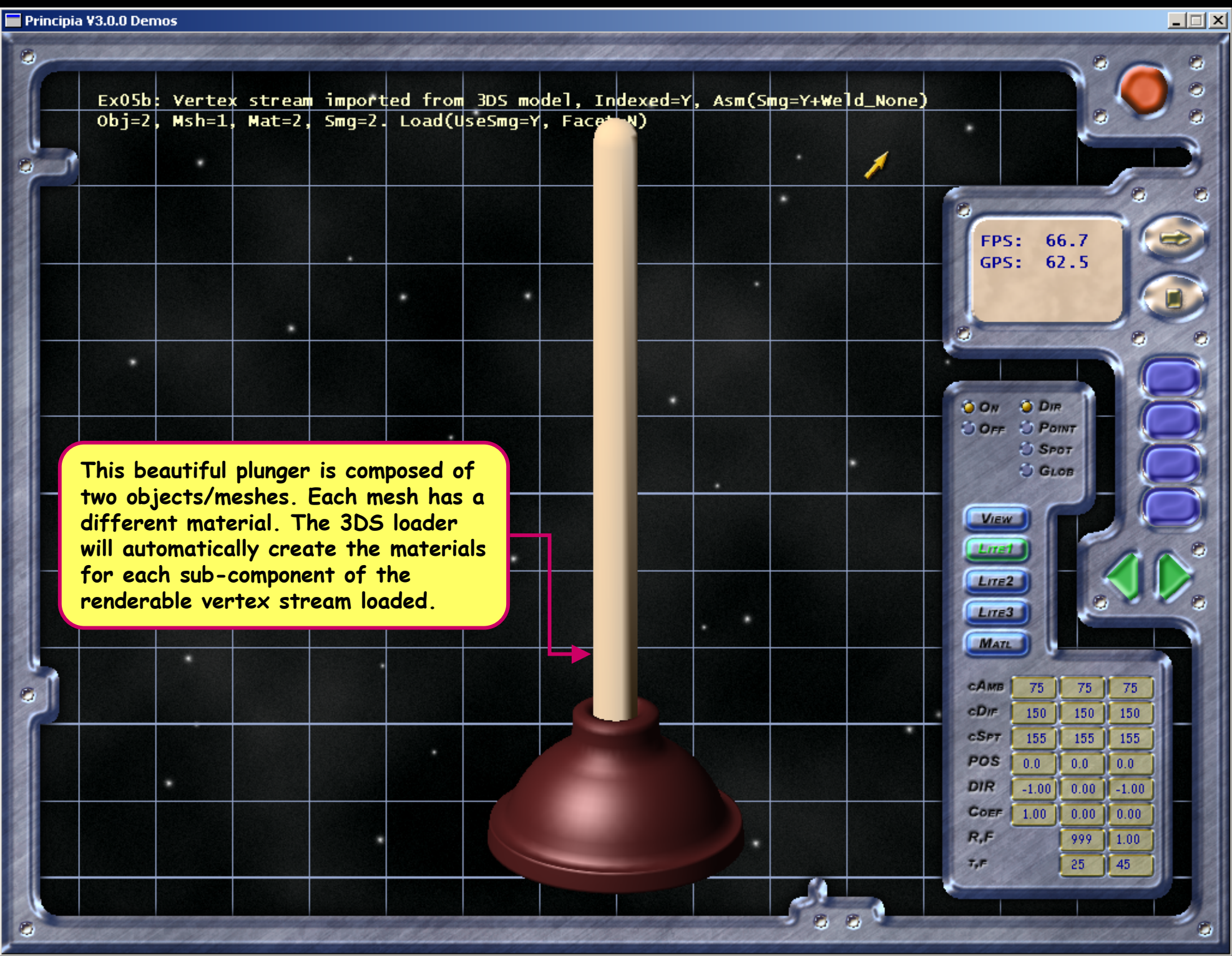
- If you deactivate or forget to consider smoothing groups in Stage1 (loading fragments), the 3DS model may not have all its sharp edges correctly captured.

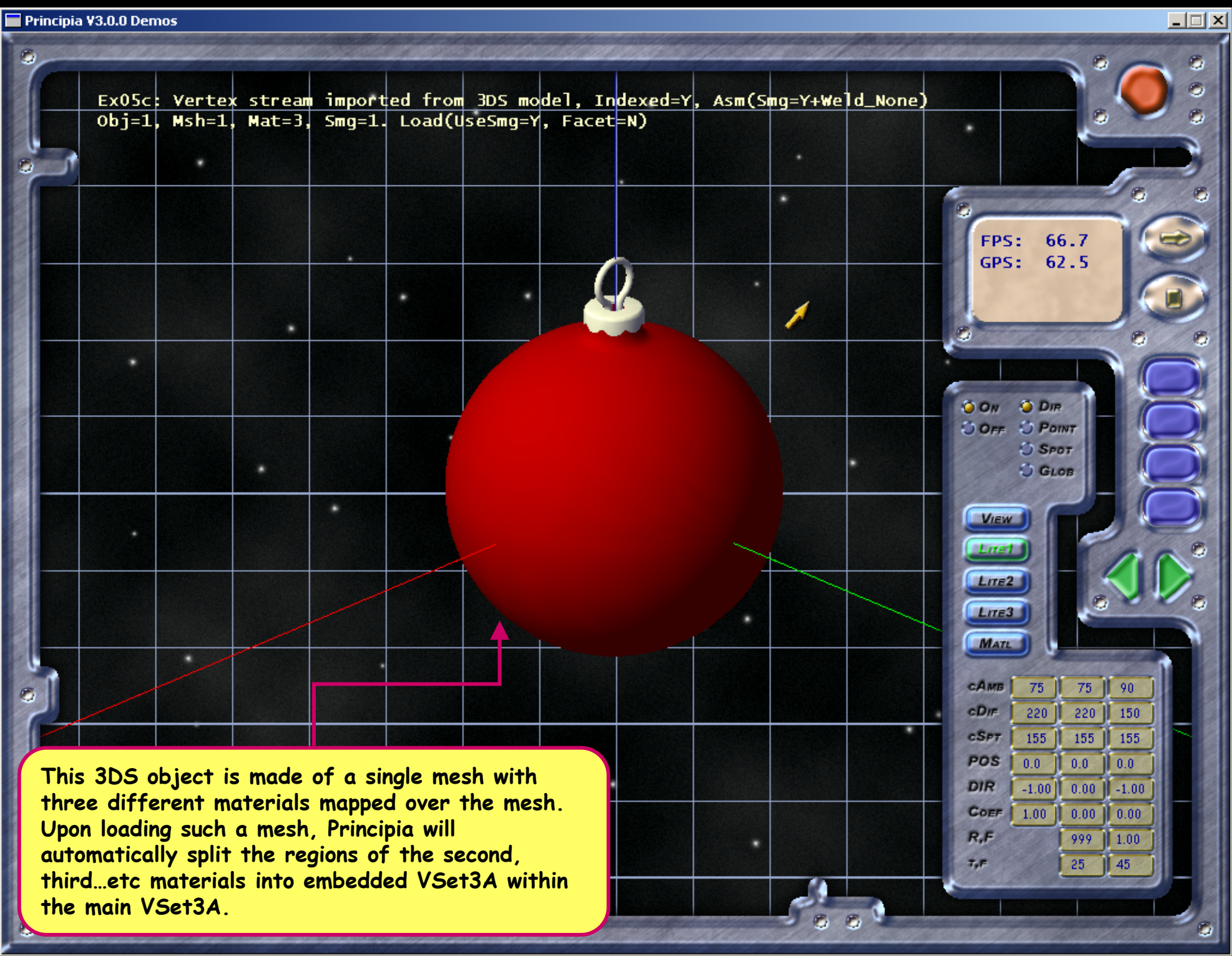
```
##define (PROC_3DSLD_EX04) as <PROCDAT_IMPORT3DS>

   #tag Transform         = "Decl(M);Mul(1.00,1.00,1.00);"
   #tag Ldo_MeshFacetize  = (   NO )
   #tag Ldo_MeshUseSmg    = (   NO )
   #tag Ldo_FragNormals   = (   NO ,         NO )
   #tag Ldo_FragMeshWeld  = (        WELD_POS  , 0.0 )
   #tag Asm_XSmg          = (   NO , WELD_NONE , 0.0 )
   #tag Asm_XMat          = (   NO , WELD_NONE , 0.0 )
   #tag Asm_XObj          = (   NO , WELD_NONE , 0.0 )
```

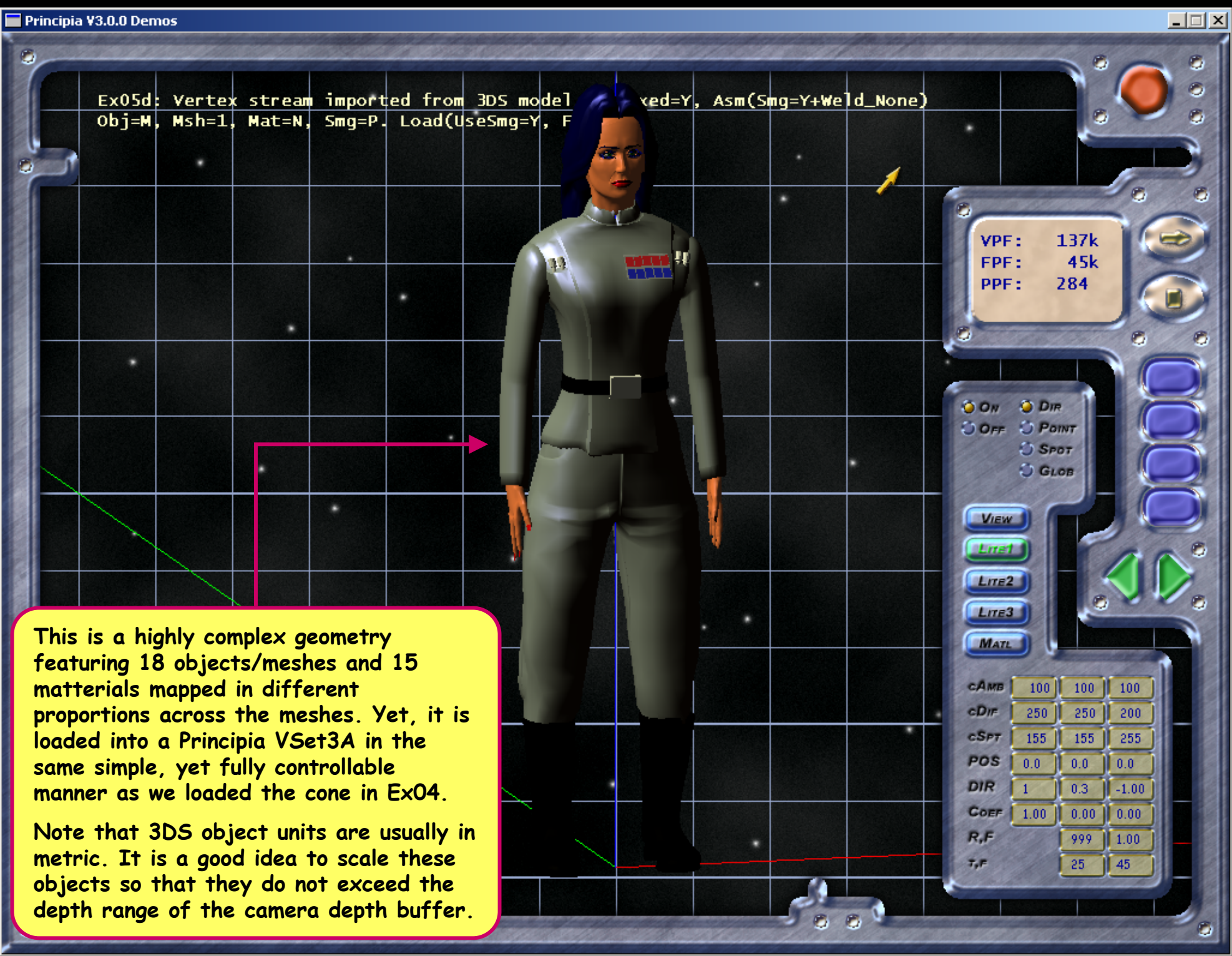
- This will be the case regardless of whether subsequent smoothing group assembly is turned on or off. The latter codes affect only how the elementary MeshUTL fragments found in the 3DS file are assembled into vertex stream primitives.




Principia V3.0.0 Demos
Ex05a: Vertex stream imported from 3DS model, Indexed=Y, Asm(Smg=Y+Weld_None)
Obj=7, Msh=1, Mat=1, Smg=0. Load(UseSmg=Y, Facet=N)
FPS: 66.7
GPS: 32.3
On
Off
Dir
Point
Spot
Glob
View
Lite1
Lite2
Lite3
Matl
cAmb 250 75 75
cDif 150 255 250
cSpt 155 155 155
POS 0.0 0.0 0.0
DIR -1.00 0.00 -1.00
Coef 1.00 0.00 0.00
R,F 999 1.00
T,F 25 45
This vertex set loaded from a multiple objects 3DS file will have embedded vertex sets (subsets) corresponding to each component. The Jeep.3ds file here, with seven objects yields a vertex set having six subsets. All subsets share the same material.


Principia V3.0.0 Demos
Ex05b: Vertex stream imported from 3DS model, Indexed=Y, Asm(Smg=Y+Weld_None)
Obj=2, Msh=1, Mat=2, Smg=2. Load(UseSmg=Y, Face
FPS: 66.7
GPS: 62.5
On
Off
Dir
Point
Spot
Glob
View
Lite1
Lite2
Lite3
Matl
cAmb 75 75 75
cDif 150 150 150
cSpt 155 155 155
POS 0.0 0.0 0.0
DIR -1.00 0.00 -1.00
Coef 1.00 0.00 0.00
R,F 999 1.00
T,F 25 45
This beautiful plunger is composed of two objects/meshes. Each mesh has a different material. The 3DS loader will automatically create the materials for each sub-component of the renderable vertex stream loaded.


Principia V3.0.0 Demos
Ex05c: Vertex stream imported from 3DS model, Indexed=Y, Asm(Smg=Y+Weld_None)
Obj=1, Msh=1, Mat=3, Smg=1. Load(UseSmg=Y, Facet=N)
FPS: 66.7
GPS: 62.5
On
Off
Dir
Point
Spot
Glob
View
Lite1
Lite2
Lite3
Matl
cAmb 75 75 90
cDif 220 220 150
cSpt 155 155 155
POS 0.0 0.0 0.0
DIR -1.00 0.00 -1.00
Coef 1.00 0.00 0.00
R,F 999 1.00
T,F 25 45
This 3DS object is made of a single mesh with three different materials mapped over the mesh. Upon loading such a mesh, Principia will automatically split the regions of the second, third…etc materials into embedded VSet3A within the main VSet3A.


Principia V3.0.0 Demos
Ex05d: Vertex stream imported from 3DS model
Obj=M, Msh=1, Mat=N, Smg=P. Load(UseSmg=Y, F
ed=Y, Asm(Smg=Y+Weld_None)
VPF: 137k
FPF: 45k
PPF: 284
On
Off
Dir
Point
Spot
Glob
View
Lite1
Lite2
Lite3
Matl
cAmb 100 100 100
cDif 250 250 200
cSpt 155 155 255
POS 0.0 0.0 0.0
DIR 1 0.3 -1.00
Coef 1.00 0.00 0.00
R,F 999 1.00
T,F 25 45
This is a highly complex geometry featuring 18 objects/meshes and 15 matterials mapped in different proportions across the meshes. Yet, it is loaded into a Principia VSet3A in the same simple, yet fully controllable manner as we loaded the cone in Ex04.
Note that 3DS object units are usually in metric. It is a good idea to scale these objects so that they do not exceed the depth range of the camera depth buffer.

D103B:Ex05 - Geometry

Objectives

- Load several more complex benchmark 3DS object
- Show how to list 3DS file components for debugging

Key Requirements

- Ex05a: Multiple objects in 3DS file
- Ex05b: Multiple materials in 3DS file (1 per object)
- Ex05c: Multiple materials per mesh in 3DS file
- Ex05d: Complex set of objects, materials and groups

Notes

- Third party 3DS files may have defects, depending on how they were created. A typical issue is the need to unify normals. It is a good idea to load them in MAX first, apply the required modifiers, and re-export them.

D103B:Ex05 - Geometry

- **To list the 3DS file chunk layout in the core dump, set the PCM_DIAGMODE_LIST3DS flag to 0x01 during compilation. This is for instance what the tree ornament contains:**

```
3DSFILE: Fna(Files_Media\Mesh_Ball.3ds)
CH:0x4D4D: Ver(3)
CH:0xDATA: Data(10193)
CH:0x3D3E: Len(10)
CH:0x100: Len(10)

CH:0x4000: Obj(Cone01)
CH:0x4100: Len(10154)
CH:0x4110: NbV(244)
CH:0x4140: NbT(244)
CH:0x4160: Mtx(     0.0000,     0.0000,11.4f)
CH:0x4120: NbF(432)
CH:0x4150: Smg(1)
CH:0xB000: Len(220)
CH:0x4D4D: Ver(3)
CH:0xDATA: Data(135125)
CH:0x3D3E: Len(10)
CH:0x100: Len(10)
CH:0x1400: Len(10)
CH:0x1420: Len(8)
CH:0x1450: Len(10)
CH:0x2100: Len(24)
CH:0x1200: Len(24)
CH:0x1300: Len(64)
CH:0x1201: Len(6)
CH:0x1460: Len(10)
CH:0x2200: Len(40)
CH:0x2300: Len(22)
CH:0x2302: Len(40)
CH:0xA052: Len(14)
CH:0xA053: Len(14)
CH:0xA084: Len(14)
CH:0xA100: Len(8)
CH:0xA087: Len(10)
CH:0xA081: Len(6)
CH:0x8000: Len(95)

. . . -> continued
```

```
CH:0xAFFF: Mat(*GLOBAL*)
   Amb:FFFFFFFF
   Dif:FFFFFFFF
   Spe:FFFFFFFF
   Pwr:0.000000
   Tex:none
   Uvp:1.000000, 1.000000, 0.000000, 0.000000
CH:0xA052: Len(14)
CH:0xA053: Len(14)
CH:0xA084: Len(14)
CH:0xA100: Len(8)
CH:0xA087: Len(10)
CH:0xA081: Len(6)
CH:0x8000: Len(95)
CH:0xAFFF: Mat(GRAY)
   Amb:FFFFFFFF
   Dif:FFFFFFFF
   Spe:FFFFFFFF
   Pwr:0.000000
   Tex:none
   Uvp:1.000000, 1.000000, 0.000000, 0.000000


. . .

CH:0x4000: Obj(0)
CH:0x4100: Len(134035)
CH:0x4110: NbV(2680)
CH:0x4140: NbT(2680)
CH:0x4111: Len(5368)
CH:0x4160: Mtx(     0.0000,     0.0000,11.4f)
CH:0x4165: Len(7)
MT: Set: Expand: (21408)
CH:0x4120: NbF(5352)
CH:0x4130: Nam(RED):NmF(4060)
CH:0x4130: Nam(GRAY):NmF(4760)
CH:0x4130: Nam(*GLOBAL*):NmF(5352)
CH:0x4150: Smg(1)
```

D103B:Ex05 - Geometry

Ex05 script with full set of loader geometry tags:

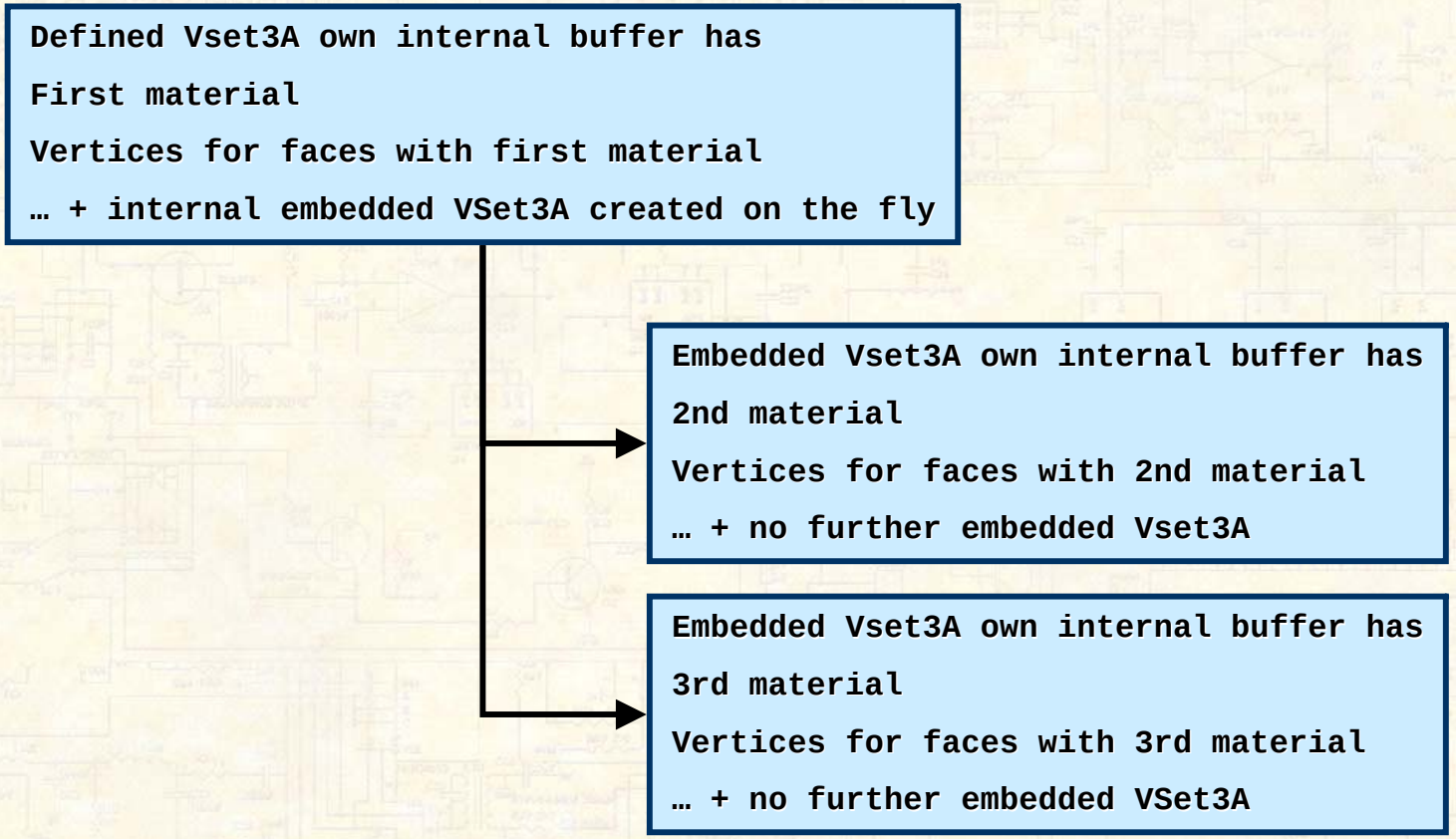
```
##define (PROC_3DSLD_EX05) as <PROCDAT_IMPORT3DS>
   @tag FilterInObjName   = (   "Name" ,  STRF_CONTAINS | STRF_CASELESS ,  0 )
   @tag FilterExObjName   = (   "Name" ,  STRF_CONTAINS | STRF_CASELESS ,  0 )
   #tag FilterInMtlName   = (    "red" ,  STRF_CONTAINS | STRF_CASELESS ,  0 )
   @tag FilterExMtlName   = (   "Name" ,  STRF_CONTAINS | STRF_CASELESS ,  0 )
   #tag FragDefiner       = (              NONE )
   #tag Transform         = "Decl(M);Mul(0.103,0.103,0.103);"
   #tag Ldo_MeshFacetize  = (   NO )
   #tag Ldo_MeshUseSmg    = (  YES )
   #tag Ldo_MeshUseMtl    = (  YES )
   #tag Ldo_FragNormals   = (   NO ,         NO )
   #tag Ldo_FragMeshWeld  = (        WELD_POS  , 0.0 )
   #tag Asm_XSmg          = (  YES , WELD_NONE , 0.0 )
   #tag Asm_XMat          = (   NO , WELD_NONE , 0.0 )
   #tag Asm_XObj          = (   NO , WELD_NONE , 0.0 )
   #tag Asm_Normals       = (   NO ,         NO )
   #tag Vgn_NamePrefix    = (             NONE )
   #tag Vgn_StripMtl      = (   NO )
   #tag Vgn_PureObjName   = (   NO )
```

Important MAX practice note: When exporting as 3DS, Max truncates many names to 10 characters.

- When just loading geometry, this does not matter. However, when loading animated or rigged character data, use short bone and biped names for correct cross-referencing.

D103B:Ex05 - Geometry

- **Resulting vertex set structure: for a single mesh, the embedded VSet cascade looks like:**



D103B – Face Culling

Culling

- Rejects back-facing polygons without rendering
- Significantly improves performance

Culling conventions

- In Principia/DirectX, front facing polygons have vertices listed in CW order due to the left-handed coordinate system.
- Some materials (such as that for line renders) disable culling. Make sure that culling is re-enabled once these materials are done, to avoid un-intentional performance hits.

Culling and face orientation from 3DS files

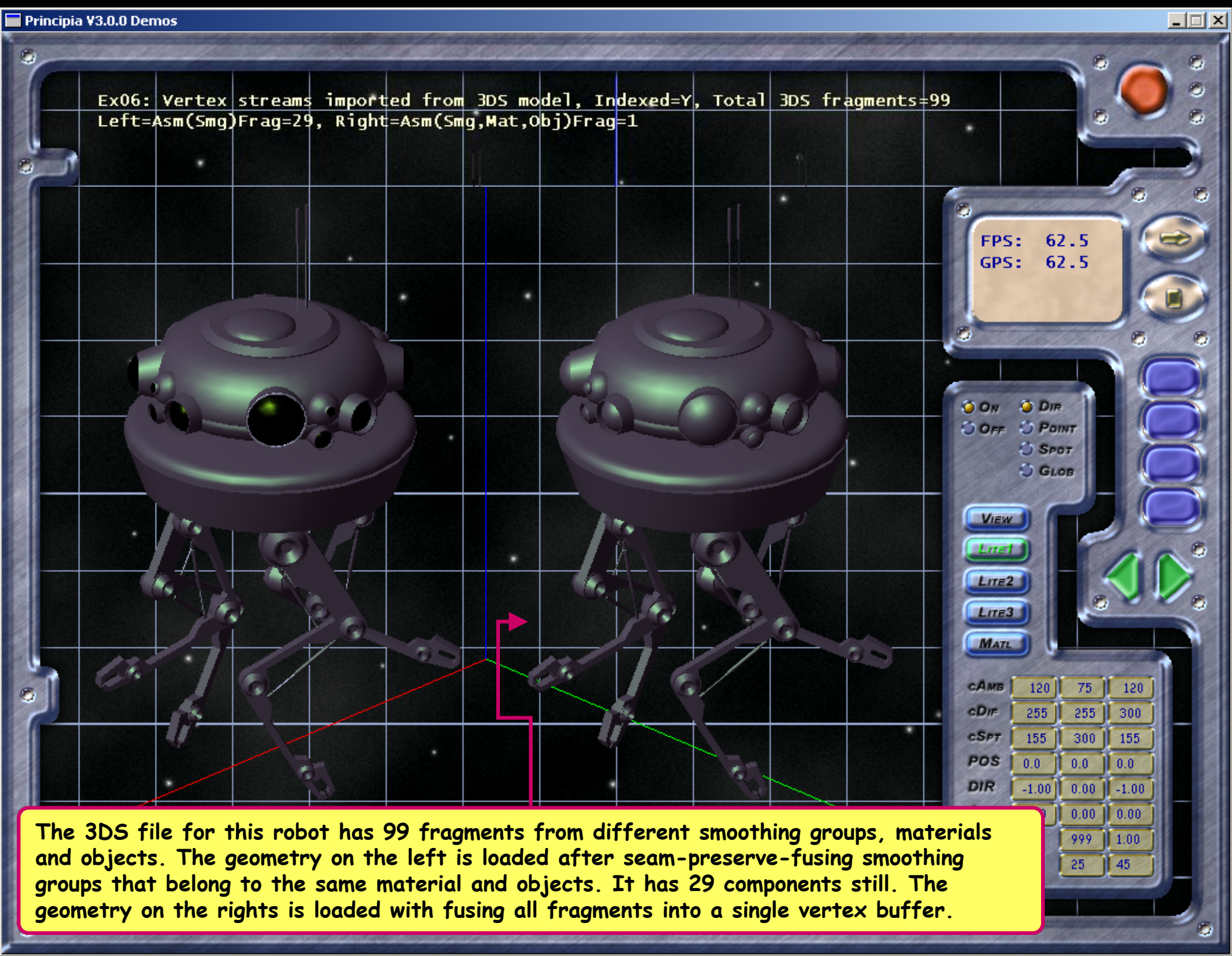
- 3DS files face index tables list the vertices listed in CCW order. Principia will automatically adjust this to conform to the CW=Front convention.

D103B – Selective Loading

- **Art production often requires specific materials or objects be excluded or solely included when loading a 3DS file.**
- **To selectively load objects from a 3DS file:**
  - Look in the 3DS diagnostic dump for the name of the object(s) and material(s) loaded during testing
  - When defining the 3DS loader procedure, use the FilterIn* or FilterEx* tags to selectively load or exclude fragments from the selected objects and/or material by name.
  - For instance, to load only the woman's left hand in this Ex05d, you must define a the 3DS loader with the following tag:

```
#tag FilterExObjName   = ( "Reference" ,  STRF_CONTAINS | STRF_CASELESS ,  0 )
```

- **Together with assembly (Ex06) the filters create a powerful capability to acquire the exact desired geometry and topology from 3DS files into the production art pipeline.**


Principia V3.0.0 Demos
Ex06: Vertex streams imported from 3DS model, Indexed=Y, Total 3DS fragments=99
Left=Asm(Smg)Frag=29, Right=Asm(Smg,Mat,Obj)Frag=1
FPS: 62.5
GPS: 62.5
On
Off
Dir
Point
Spot
Glob
View
Lite1
Lite2
Lite3
Matl
cAmb 120 75 120
cDif 255 255 300
cSpt 155 300 155
POS 0.0 0.0 0.0
DIR -1.00 0.00 -1.00
0.00 0.00
999 1.00
25 45
The 3DS file for this robot has 99 fragments from different smoothing groups, materials and objects. The geometry on the left is loaded after seam-preserve-fusing smoothing groups that belong to the same material and objects. It has 29 components still. The geometry on the rights is loaded with fusing all fragments into a single vertex buffer.

D103B:Ex06 - Geometry

Objectives

- Show fragment assembly when loading 3DS files
- Show saving multi-component vertex streams

Key Requirements

- Load raw 3DS file with 105 fragments
- Exclude the reference grid from the 3DS file
- Geometry 1 with smoothing groups assembly only
- Geometry 2 with all fragments assembled

Notes

- It is a common performance-enhancing production practice to bake materials and aggregate primitives.
- The Principia 3DS loader features powerful options to assemble 3DS fragments in any desired manner.

D103B:Ex06 - Geometry

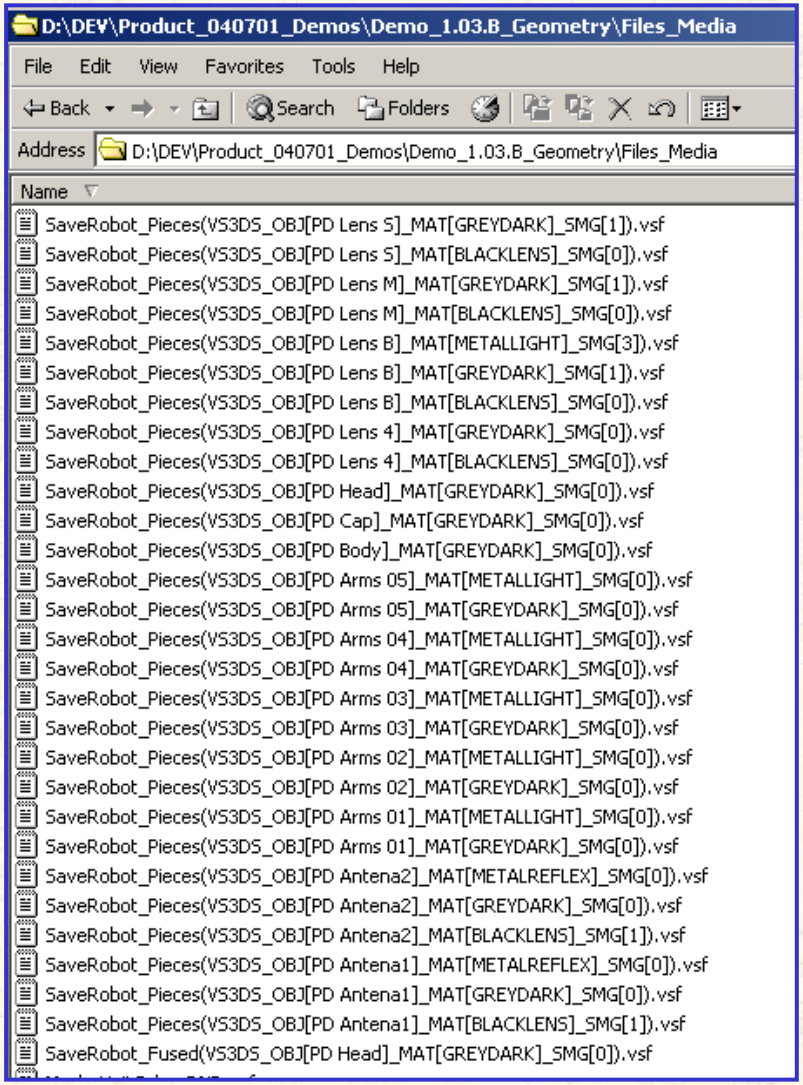
- **Implementation: geometry loaded in pieces and saved as individual fragments. Note the exclusion of an unwanted object from the 3DS file.**

```
##define (PROC_3DSLD_EX06_PIECES) as <PROCDAT_IMPORT3DS>
   #tag FilterExObjName   = ( "Reference" ,  STRF_CONTAINS | STRF_CASELESS ,  0 )
   #tag Transform         = "Decl(M);Mul(0.0003,0.0003,0.0003);"
   #tag Ldo_MeshFacetize  = (   NO )
   #tag Ldo_MeshUseSmg    = (  YES )
   #tag Ldo_FragNormals   = (   NO ,         NO )
   #tag Ldo_FragMeshWeld  = (         WELD_POS , 0.0 )
   #tag Asm_XSmg          = (  YES , WELD_NONE , 0.0 )
   #tag Asm_XMat          = (   NO , WELD_NONE , 0.0 )
   #tag Asm_XObj          = (   NO , WELD_NONE , 0.0 )
   #tag Vgn_NamePrefix    = (            NONE )


##define (G_EX06_PIECES) as <VERTEXSET_3A>
   #tag File      =  "Files_Media\Mesh_Droid.3ds"
   #tag Definer   =  (    VTD_FVF_FILE )
   #tag LoadProc  =  ( PROC_3DSLD_EX06_PIECES )
   #tag Indexed   =  (              YES )
   #tag LoadNow   =  (                1 )

##save (G_EX06_PIECES) in ("Files_Media\SaveRobot_Pieces.vsf")
```

- **Note that is customary to fuse fragments that differ only by smoothing groups with *an option to preserve the sharp seam*. This greatly reduces fragment count and retains the geometry fully.**

D103B:Ex06 - Geometry

- Best practice: Load 3DS, save as VSF, reload VSF.
- When saved, multi-component vertex streams loaded from 3DS will store each component in a separately named VSF file that reflects its originating 3DS object, material ...etc.
- In production, these components can be loaded as individual meshes and organized in the rendering sequence in a more efficient manner than if we were loading a single vertex stream with sub-components.



D103B:Ex06 - Geometry

- **Implementation: geometry fused in a single vertex stream (all ASM flags set to YES).**

```
##define (PROC_3DSLD_EX06_FUSED) as <PROCDAT_IMPORT3DS>
   #tag FilterExObjName   = ( "Reference" ,  STRF_CONTAINS | STRF_CASELESS ,  0 )
   #tag Transform         = "Decl(M);Mul(0.0003,0.0003,0.0003);"
   #tag Ldo_MeshFacetize  = (    NO )
   #tag Ldo_MeshUseSmg    = (   YES )
   #tag Ldo_FragNormals   = (    NO ,         NO )
   #tag Ldo_FragMeshWeld  = (          WELD_POS , 0.0 )
   #tag Asm_XSmg          = (   YES , WELD_NONE , 0.0 )
   #tag Asm_XMat          = (   YES , WELD_NONE , 0.0 )
   #tag Asm_XObj          = (   YES , WELD_NONE , 0.0 )
    #tag Vgn_NamePrefix    = (            NONE )


##define (G_EX06_FUSED) as <VERTEXSET_3A>
   #tag File      =  "Files_Media\Mesh_Droid.3ds"
   #tag Definer   =  (    VTD_FVF_FILE )
   #tag LoadProc  =  ( PROC_3DSLD_EX06_FUSED )
   #tag Indexed   =  (             YES )
   #tag LoadNow   =  (               1 )
```

- **Note that the resulting stream has lost its materials definition (sharp edges are still correctly retained in assembly with WELD_NONE)**

D103B:Ex06 - Geometry

- **Assembly sequence and priority:**
  - AsmXSmg: Customarily set on. Assembles fragments with different smoothing groups and same object/material. Seam preservation is controlled by the WELD_* tag argument.
  - AsmXMat: Assembles fragments with different materials but same object/mesh. This flag assumes that smoothing groups have been already assembled, and does not preserve them.
  - AsmXObj: Assembles fragments across different objects and meshes in the 3DS file. Depending on the AsmXMat setting, this will also fuse all materials, or assemble fragments along identical material groups. This flag also assumes that smoothing groups have been assembled.
- **To weld a complex 3DS structure into a single vertex stream, turn on all three flags on and set WELD_NONE to preserve seam.**

D103B:Ex06 - Geometry

- **Advanced 3DS data loading preview:**
  - The 3DS vertex format is limited to positions and texture coordinates. Per-vertex color, weights, indices ...etc. cannot be retained in 3DS files.
  - Principia features several specialized procedures to extract such vertex data directly from MAX or other modeling programs, so that you can import rigged characters and complex artwork directly into Principia.
  - Some shaders require per-vertex tangent, binormal, curvature...etc. data. The FX_Import3DS loader can add full per-vertex tangent space data to the geometry being loaded. It can also store the geometry in UTL and VSF formats for loading into other Principia geometry procedures.
- **These advanced 3DS/Max loading techniques will be covered in subsequent demos.**

Principia V3.0.0 Demos

Ex07a: Flat geometry renders with the programmable pipeline.
Flat image sprite placed by KX, unit rectangle, design choice of KX+VS+PS

FPS: 66.7
GPS: 62.5

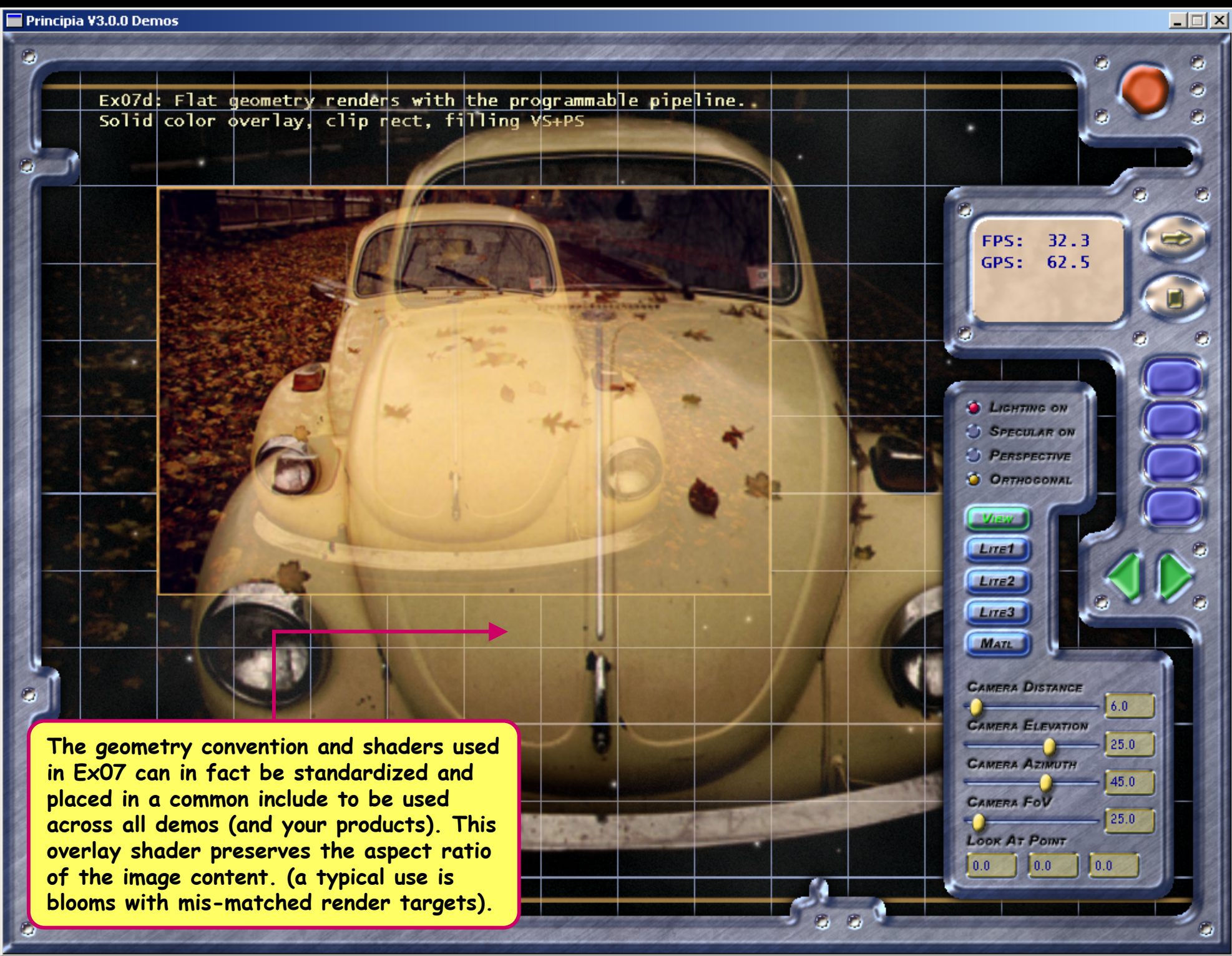
Lighting on
Specular on
Perspective
Orthogonal

View
Lite1
Lite2
Lite3
Matl

Camera Distance 6.0
Camera Elevation 25.0
Camera Azimuth 45.0
Camera FoV 25.0
Look At Point 0.0 0.0 0.0

Flat sprites still live …in fact it is pretty common to insert flat images in the 3D rendering stream. Ex07 shows how to manage geometry to render flat images with shaders at a desired size and screen position.


Principia V3.0.0 Demos
Ex07d: Flat geometry renders with the programmable pipeline.
Solid color overlay, clip rect, filling VS+PS
FPS: 66.7
GPS: 62.5
Lighting on
Specular on
Perspective
Orthogonal
View
Lite1
Lite2
Lite3
Matl
Camera Distance
6.0
Camera Elevation
25.0
Camera Azimuth
45.0
Camera FoV
25.0
Look At Point
0.0
0.0
0.0
Overlays are a special form of flat renders, covering the entire render target with pixel content. Overlays are also very common and essential for some SFX.


Principia V3.0.0 Demos
Ex07d: Flat geometry renders with the programmable pipeline.
Solid color overlay, clip rect, filling VS+PS
FPS: 32.3
GPS: 62.5
Lighting on
Specular on
Perspective
Orthogonal
View
Lite1
Lite2
Lite3
Matl
Camera Distance
6.0
Camera Elevation
25.0
Camera Azimuth
45.0
Camera FoV
25.0
Look At Point
0.0
0.0
0.0
The geometry convention and shaders used in Ex07 can in fact be standardized and placed in a common include to be used across all demos (and your products). This overlay shader preserves the aspect ratio of the image content. (a typical use is blooms with mis-matched render targets).

Principia V3.0.0 Demos

Ex07d: Flat geometry renders with the programmable pipeline.
Solid color overlay, clip rect, filling VS+PS

FPS: 62.5
GPS: 66.7

Lighting on
Specular on
Perspective
Orthogonal

View
Lite1
Lite2
Lite3
Matl

Camera Distance 6.0
Camera Elevation 25.0
Camera Azimuth 45.0
Camera FoV 25.0
Look At Point 0.0 0.0 0.0

Another type of render target overlay is the solid ARGB fill and blend. There are many possible shader-based designs for rendering flat geometry. The common include is but one of them.

The point of Ex07 is to help you understand input and output geometry with the programmable pipeline. Oh, why do I need shaders to render a flat image? Simple – endless creative possibilities for FX, and the fixed function pipeline is going away…

D103B:Ex07 - Geometry

Objectives

- Show input/output geometry convention for rendering flat images and render target overlays with shaders.

Requirements and Implementation

- Render flat image (sprite) with size and location encapsulated by the kinex world transform.
- Render a flat overlay that fully covers the viewport regardless of any distortion involved.
- Render a flat overlay that covers the viewport while preserving the aspect ratio of the overlay image.

Notes:

- Viewport overlays are common in many SFX, while many commercial products still use flat images. Ex07 shows just one design option of the many that are possible if you understand GPU geometry processing.

D103B:Ex07 - Geometry

Geometry in programmable rendering:

- Input geometry can be in any units you want
- Geometry placement parameters (display position and scale) can be passed to the shader in any way you want (camera transforms, kinex matrices, variables bound to shader registers ...etc).
- Vertex shader output is in clipped (homogeneous) space, which covers the [-1,1] XY [01] Z range. The depth Z can be used to control output stacking.
- The vertical clip space position runs from top to bottom, which requires inverting input that is assumed positive from bottom to top.

Geometry in fixed pipeline (Chapter 11):

- Untransformed input geometry subject to same output requirements as with the programmable pipeline. The input will be processed using the fixed pipeline vertex TNL.
- In addition, can pass transformed geometry with exact pixel position coordinates without any vertex processing.

D103B:Ex07 - Geometry

Design choices for Ex07 and throughout demos:

- Flat images: input geometry is a unit rectangle (XY in [01] with UV in [01] along L-R and B-T). The size and location of the image rendered using this geometry is specified in the world matrix of the kinex used to render this geometry.
- Overlays: input geometry is a rectangle covering the full clip space extent (XY in [-1,1] with UV in [01] along L-R and T-B). The V texcoord is inverted to fit the screen T-B pixel coordinate convention.
- We use the standard register bindings in SHD3D to pass viewport (c50) and transform data (c0) to the shaders that will render flat images and overlays.

Common encapsulation:

- Because flat image and overlays are used in many places, the common geometries and several typical shaders are included in Common_FlatRenders.cfg.

D103B:Ex07 - Geometry

Standard “flat renders” include geometries

- Use the two-triangle version for direct image renders with the standard include shaders.
- Use the high-triangle version for displacement and texture coordinate effects with custom shaders.

```
@@Flat rectangle for sprites; UnitXY, two triangles; U-L-R, V-B-T; POS:3F,NORM:3F,CD:CLR,TEX:2F; flat white color
##define (G_STD_FLATSPRITE) as <VERTEXSET_3A>
   #tag File      =  "../Demo_Common/Files_Media/Meshes/Mesh_FlatRect_UnitSquare[2P-FFFFFFFF-RNDT].vsf"
   #tag Definer   =  ( VTD_DCL_FILE )

@@Flat rectangle for overlays; ClipXY; two triangles; U-L-R, V-T-B; POS:3F,NORM:3F,CD:CLR,TEX:2F; flat white color
##define (G_STD_FLATOVERLAY) as <VERTEXSET_3A>
   #tag File      =  "../Demo_Common/Files_Media/Meshes/Mesh_FlatRect_ClipOverlay[2P-FFFFFFFF-RNDT].vsf"
   #tag Definer   =  ( VTD_DCL_FILE )



@@Flat rectangle for sprites; UnitXY, 512 triangles; U-L-R, V-B-T; POS:3F,NORM:3F,CD:CLR,TEX:2F; flat white color
##define (G_STD_RICHSPRITE) as <VERTEXSET_3A>
   #tag File      =  "../Demo_Common/Files_Media/Meshes/Mesh_FlatRect_UnitSquare[LP-FFFFFFFF-RNDT].3ds"
   #tag Definer   =  ( VTD_DCL_FILE )

@@Flat rectangle for overlays; ClipXY; 512 triangles; U-L-R, V-T-B; POS:3F,NORM:3F,CD:CLR,TEX:2F; flat white color
##define (G_STD_RICHOVERLAY) as <VERTEXSET_3A>
   #tag File      =  "../Demo_Common/Files_Media/Meshes/Mesh_FlatRect_ClipOverlay[LP-FFFFFFFF-RNDT].3ds"
   #tag Definer   =  ( VTD_DCL_FILE )
```

D103B:Ex07 - Geometry

- The unit XY square for “flat sprite” renders
  - Normal and color data is included but not used
  - Note that the UV coordinates are not flipped, hence Y is positive upwards. The display inversion will be done in the vertex shader.

```
TEXT
VSF3
UNIT RECTANGLE FOR RENDERING FLAT GEOMETRY SPRITES WITH FULL BASE COLOR (NONINVERTED V)
@@ Vertex format specification
POS:3F, NORM:3F, COLOR:COLOR, TEX:2F
@@ Primitive type
4
@@ Indexed flag
0
@@ Reserved bootstrap section
0
@@ Total number of vertices and indices
36
0
@@ Number of frames
1
@@ Frame#0: Primitive first vertex index, primitive count, number of vertices
0  2  6
@@ Frame#0: Locus of first frame in index array and index count per frame
0  0
@@ Vertex data Tr01 for CCW cull
1.00  1.00  1.00    0.0  0.0  1.0    FFFFFFFF   1.0  1.0
0.00  0.00  1.00    0.0  0.0  1.0    FFFFFFFF   0.0  0.0
1.00  0.00  1.00    0.0  0.0  1.0    FFFFFFFF   1.0  0.0
@@ Vertex data Tr02 for CCW cull
1.00  1.00  1.00    0.0  0.0  1.0    FFFFFFFF   1.0  1.0
0.00  1.00  1.00    0.0  0.0  1.0    FFFFFFFF   0.0  1.0
0.00  0.00  1.00    0.0  0.0  1.0    FFFFFFFF   0.0  0.0
```

D103B:Ex07 - Geometry

- **The [-1,1] square for RT overlay renders**
  - **Overlays widely used for many SFX (blooms, shadow volumes, RT blends...etc)**
  - **Geometry supports three overlay shaders (fill with image, fill with preserved-AR image, fill with ARGB).**

```
TEXT
VSF3
UNIT RECTANGLE FOR RENDERING FLAT GEOMETRY SPRITES WITH FULL BASE COLOR (INVERTED V)
@@ Vertex format specification
POS:3F, NORM:3F, COLOR:COLOR, TEX:2F
@@ Primitive type
4
@@ Indexed flag
0
@@ Reserved bootstrap section
0
@@ Total number of vertices and indices
36
0
@@ Number of frames
1
@@ Frame#0: Primitive first vertex index, primitive count, number of vertices
0  2  6
@@ Frame#0: Locus of first frame in index array and index count per frame
0  0
@@ Vertex data Tr01 for CCW cull
1.00  1.00  1.00    0.0  0.0  1.0    FFFFFFFF   1.0  0.0
1.00  -1.0  1.00    0.0  0.0  1.0    FFFFFFFF   1.0  1.0
-1.0  -1.0  1.00    0.0  0.0  1.0    FFFFFFFF   0.0  1.0
@@ Vertex data Tr02 for CCW cull
1.00  1.00  1.00    0.0  0.0  1.0    FFFFFFFF   1.0  0.0
-1.0  -1.0  1.00    0.0  0.0  1.0    FFFFFFFF   0.0  1.0
-1.0  1.00  1.00    0.0  0.0  1.0    FFFFFFFF   0.0  0.0
```

D103B:Ex07 - Geometry

- **Standard include vertex shaders for rendering flat geometry (the pixel shaders are trivial).**

```
@@VS for flat sprites, c0 matrix encodes sprite size and position, c50 holds viewport size
@@Inputs expect std unit geomery, outputs flat white color per vertex, alpha encoded in texture
##define (VS_IMG_SPRITE) as <VSHADER_3A>
   #tag File      =  "../Demo_Common/Files_Media/Shaders/VS_Img_UnitSprite.shd"
   #tag ShaderFcn =  "VS_Img_UnitSprite"
   #tag ShaderAsm =  (           NO )
   #tag ShaderVs  =  (          1.1 )

@@VS for image overlay with texture alpha channel. Input expects std overlay geometry.
@@This vertex shader will stretch the image to fit the viewport size
##define (VS_IMG_OVERLAY_FULLIMG) as <VSHADER_3A>
   #tag File      =  "../Demo_Common/Files_Media/Shaders/VS_Img_Overlay_Full.shd"
   #tag ShaderFcn =  "VS_Img_Overlay_Full"
   #tag ShaderAsm =  (           NO )
   #tag ShaderVs  =  (          1.1 )

@@VS for image overlay with texture alpha channel. Input expects std overlay geometry.
@@This vertex shader will adjust output so that the image fits the viewport but keeps its AR.
@@Viewport size is expected in c50, image size in c0 scale matrix.
##define (VS_IMG_OVERLAY_KEEPIMG) as <VSHADER_3A>
   #tag File      =  "../Demo_Common/Files_Media/Shaders/VS_Img_Overlay_Keep.shd"
   #tag ShaderFcn =  "VS_Img_Overlay_Keep"
   #tag ShaderAsm =  (           NO )
   #tag ShaderVs  =  (          1.1 )

@@VS for image overlay with flat ARGB passed by shader register variable.
@@Input expects std overlay geometry. Color expectecd in c29.
##define (VS_IMG_OVERLAY_RGBA) as <VSHADER_3A>
   #tag File      =  "../Demo_Common/Files_Media/Shaders/VS_Img_Overlay_Rgba.shd"
   #tag ShaderFcn =  "VS_Img_Overlay_Rgba"
   #tag ShaderAsm =  (           NO )
   #tag ShaderVs  =  (          1.1 )
```

D103B:Ex07 - Geometry

- **Vertex shader to render flat “sprite” at XY with desired WH dimension carried by the kinex.**

```
//--------------------------------------------------------------------------//
//  Shader data registers, variables and static constants                   //
//--------------------------------------------------------------------------//
##include "..\Demo_Common\Files_Media\Shaders\VS_Std_Includes.shd"

struct VS_INPUT
{   // Inbound vertex buffer stream
    float3 vPosition        : POSITION ;
    float2 vTexCoords       : TEXCOORD0;
};

struct VS_OUTPUT
{   // Extant processed vertex stream
    float4 vPosition        : POSITION ;
    float4 vDiffuse         : COLOR0   ;
    float2 vTexCoord0       : TEXCOORD0;
};

//--------------------------------------------------------------------------//
//  Shader principal operations                                             //
//--------------------------------------------------------------------------//
VS_OUTPUT VS_Img_UnitSprite (const VS_INPUT In) { VS_OUTPUT Out = (VS_OUTPUT) 0;

    /* Kinex carries sprite position and size. Using viewport */
    /* dimensions, transform the data into homogeneous coords */
    Out.vPosition    = mul(float4(In.vPosition,1.0f), mWrld);
    Out.vPosition.x  = (Out.vPosition.x + vPort.x) * 2.0 / (vPort.z - vPort.x) - 1.0;
    Out.vPosition.y  = 1.0 - (Out.vPosition.y + vPort.y) * 2.0 / (vPort.w - vPort.y);
    Out.vPosition.z  = 0.0;
    Out.vPosition.w  = 1.0;

    /* Use only base texture with flat white diffuse */
    Out.vDiffuse     = float4( 1.0, 1.0, 1.0, 1.0);
    Out.vTexCoord0   = In.vTexCoords;

return Out; }
//--------------------------------------------------------------------------//
//  Shader principal operations                                             //
//--------------------------------------------------------------------------//
```

D103B:Ex07 - Geometry

- **Vertex shader to overlay render target with an image, stretching it fully to fill RT.**
  - **The position transform assumes using the overlay clip space [-1,1] mesh with inverted V-texcoord.**

```
//-----------------------------------------------------------------------------//
//  Shader data registers, variables and static constants                      //
//-----------------------------------------------------------------------------//
##include "..\Demo_Common\Files_Media\Shaders\VS_Std_Includes.shd"

struct VS_INPUT
{   // Inbound vertex buffer stream
    float3 vPosition        : POSITION ;
    float2 vTexCoords       : TEXCOORD0;
};

struct VS_OUTPUT
{   // Extant processed vertex stream
    float4 vPosition        : POSITION ;
    float4 vDiffuse         : COLOR0   ;
    float2 vTexCoord0       : TEXCOORD0;
};

//-----------------------------------------------------------------------------//
//  Shader principal operations                                                //
//-----------------------------------------------------------------------------//
VS_OUTPUT VS_Img_Overlay_Full (const VS_INPUT In)  {  VS_OUTPUT Out = (VS_OUTPUT) 0;

    /* Full coverage of clip space */
    Out.vPosition.x  = (In.vPosition.x);
    Out.vPosition.y  = (In.vPosition.y);
    Out.vPosition.z  = 0.0;
    Out.vPosition.w  = 1.0;

    /* Use only base texture with flat white diffuse */
    Out.vDiffuse     = float4( 1.0, 1.0, 1.0, 1.0);
    Out.vTexCoord0   = In.vTexCoords;

return Out; }
//-----------------------------------------------------------------------------//
//  Shader principal operations                                                //
//-----------------------------------------------------------------------------//
```

D103B:Ex07 - Geometry

- Vertex shader to overlay and keep aspect ratio.
  Note the use of viewport and source image size.

TO BE FINALIZED

D103B:Ex07 - Geometry

- **Vertex shader to fill the overlay with solid color**
  - **Commonly used in many shadow and transition FX**
  - **Expects color to be passed in register c29 by material**

```
//--------------------------------------------------------------------------//
//  Shader data registers, variables and static constants                   //
//--------------------------------------------------------------------------//
##include "..\Demo_Common\Files_Media\Shaders\VS_Std_Includes.shd"
float4   OverColor : register (c29); //Api: Overlay ARGB color

struct VS_INPUT
{   // Inbound vertex buffer stream
    float3 vPosition        : POSITION ;
    float2 vTexCoords       : TEXCOORD0;
};

struct VS_OUTPUT
{   // Extant processed vertex stream
    float4 vPosition        : POSITION ;
    float4 vDiffuse         : COLOR0   ;
    float2 vTexCoord0       : TEXCOORD0;
};

//--------------------------------------------------------------------------//
//  Shader principal operations                                             //
//--------------------------------------------------------------------------//
VS_OUTPUT VS_Img_Overlay_Rgba (const VS_INPUT In)  {  VS_OUTPUT Out     = (VS_OUTPUT) 0;

    /* Full coverage of clip space */
    Out.vPosition.x  = (In.vPosition.x);
    Out.vPosition.y  = (In.vPosition.y);
    Out.vPosition.z  = 0.0;
    Out.vPosition.w  = 1.0;

    /* Use only base texture with flat white diffuse */
    Out.vDiffuse     = OverColor;
    Out.vTexCoord0   = In.vTexCoords;

return Out; }
//--------------------------------------------------------------------------//
//  Shader principal operations                                             //
//--------------------------------------------------------------------------//
```

D103B:Ex07 - Geometry

- **Rendering setup for a “flat image” sprite with the common include shaders**
  - **Kinex must pass insert position and image size on screen in pixels, relative to upper left corner.**
  - **Typical design used to render flat screen-space primitives as part of the world rendering sequence.**

```
##define (M_EX07_SPRITE) as <MATERIAL_3A>
   #tag VShader   =  (             VS_IMG_SPRITE )
   #tag PShader   =  (          PS_IMG_TEXOVERLAY )
   #tag TX        =  (           S_EX07_IMG ,  0 )

##define (R_EX07_SPRITE) as <MESH_3A>
   #tag Component =  (            0 ,  M_EX07_COMPREFIX ,                  - )
   #tag Component =  (            0 ,     M_EX07_SPRITE ,    G_STD_FLATSPRITE )

##define (O_EX07_SPRITE) as <GENOBJECT_3A>
   #tag Construct =  (  "Decl(M); Mul(512.0, 341.0, 0); Mov(128.0, 128.0, 0); Set(tM,0);" ,   R_EX07_SPRITE ,       NONE )
```

D103B:Ex07 - Geometry

- **Rendering setup for the RT overlays by an image. Note that the aspect-ratio preserving set-up requires the image size to be passed in Mo.**

```
##define (M_EX07_FULLOVER) as <MATERIAL_3A>
   #tag VShader    =  (      VS_IMG_OVERLAY_FULLIMG )
   #tag PShader    =  (           PS_IMG_TEXOVERLAY )
   #tag TX         =  (             S_EX07_OVR ,  0 )

##define (R_EX07_FULLOVER) as <MESH_3A>
   #tag Component =  (            0 ,  M_EX07_COMPREFIX ,                    - )
   #tag Component =  (            0 ,   M_EX07_FULLOVER ,    G_STD_FLATOVERLAY )

##define (O_EX07_FULLOVER) as <GENOBJECT_3A>
   #tag Construct =  (  "Decl(M); Set(tM,0);" ,   R_EX07_FULLOVER ,       NONE )


##define (M_EX07_ADJOVER) as <MATERIAL_3A>
   #tag VShader    =  (      VS_IMG_OVERLAY_KEEPIMG )
   #tag PShader    =  (           PS_IMG_TEXOVERLAY )
   #tag TX         =  (             S_EX07_OVR ,  0 )

##define (R_EX07_ADJOVER) as <MESH_3A>
   #tag Component =  (            0 ,  M_EX07_COMPREFIX ,                    - )
   #tag Component =  (            0 ,    M_EX07_ADJOVER ,    G_STD_FLATOVERLAY )

##define (O_EX07_ADJOVER) as <GENOBJECT_3A>
   #tag Construct =  (  "Decl(M); Mul(512.0, 341.0, 0); Set(tM,0);" ,   R_EX07_ADJOVER ,       NONE )
```

D103B:Ex07 - Geometry

- **Rendering setup for a constant ARGB color overlay. The material passes the desired color to the common include vertex shader.**

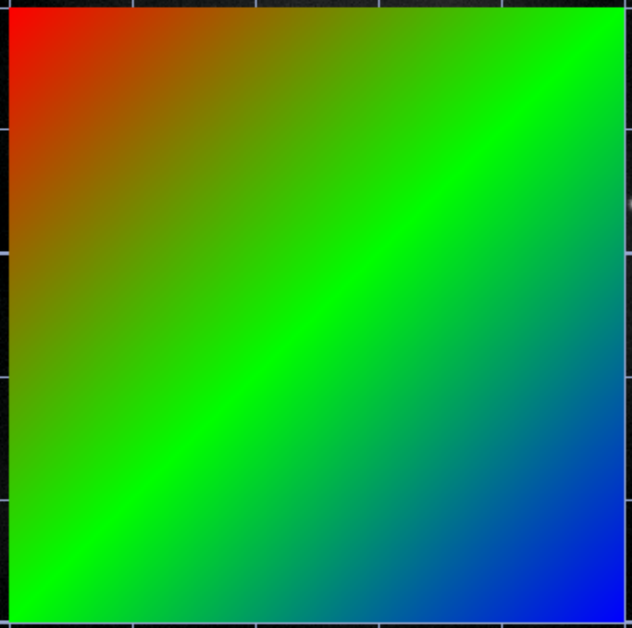
```
##define <SYS_VARIABLE>  =  (  V_COLORFILL ,   VARTYPE_FSR  , 0.5 , 0.5 , 0.5 , 0.5 )

##define (M_EX07_RGBAOVER) as <MATERIAL_3A>
   #tag VShader   =  (         VS_IMG_OVERLAY_RGBA )
   #tag PShader   =  (           PS_IMG_CLROVERLAY )
   #tag VsVar     =  (   29 ,          V_COLORFILL )

##define (R_EX07_RGBAOVER) as <MESH_3A>
   #tag Component =  (            0 ,  M_EX07_COMPREFIX ,                    - )
   #tag Component =  (            0 ,    M_EX07_RGBAOVER ,   G_STD_FLATOVERLAY )

##define (O_EX07_RGBAOVER) as <GENOBJECT_3A>
   #tag Construct =  (  "Decl(M); Set(tM,0);" ,   R_EX07_RGBAOVER ,       NONE )
```

- **Remember to put this in your script. We will use these “standard” flat render tools a lot…**

```
##parse "../Demo_Common/Files_Scripts/Common_FlatRenders.cfg"
```

Principia V3.0.0 Demos

Ex07: Flat pseudo2D geometry, fixed-function pipeline render.
Geometry units=transformable pixels. Camera settings=Pseudo2D



FPS: 62.5
GPS: 62.5

Lighting on
Specular on
Perspective
Orthogonal

View
Lite1
Lite2
Lite3
Matl

Camera Distance 6.0
Camera Elevation 25.0
Camera Azimuth 45.0
Camera FoV 25.0
Look At Point 0.0 0.0 0.0

Great many applications still render flat images as part of their graphic output. Principia provides many ways to output flat images. This demo shows one method that recaps how to specify geometry and camera settings.

This basic rectangle is made out of two triangle primitives, specified in pixel units. The camera settings are made to emulate a 2D display. The object is positioned on the screen using a kinex with pixel translation units, and rendered using the fixed function pipeline.

Remember - The (0,0) origin is the center of the viewport, and lighting has to be turned off because there is no normal data in this geometry. Pseudo2D objects can be lit if normals are provided.

D103B:Ex08 - Geometry

- **Objectives**
  - Show how set up transformed geometry for emulating 2D “blit” renders using the fixed function pipeline.
- **Requirements and Implementation**
  - Draw a single rectangle 320 pixels to the side at pixel position (484,256) on the screen. Use a geometry that is sized in pixels for that purpose.
  - Be able to reposition this rectangle, turn it around, scale it ...etc. without changing its vertex coordinates. Use a kinex sized in pixels for that purpose.
  - Set camera to emulate 2D flat projection to viewport and turn GPU lighting off.
- **Notes:**
  - Legacy flat geometry rendered in the fixed function pipeline approach enjoys quite a broad use still.

D103B:Ex08 - Geometry

- **Principia offers many pathways for rendering flat images, with or without shaders:**
  - **Manually setting up flat geometry and camera settings that emulate 2D rendering. The geometry can be:**
    - **Sized in pixels (as in the present Ex08)**
    - **Sized in absolute viewport units (as in Ex07)**
  - **Using point vertex primitives. Note that these are being discontinued in DirectX10.**
  - **Using Principia objects of the <Sprite> class which automatically set up each object world transform so that it renders as a camera-facing tile in a 3D world.**
  - **Using vertex buffers with transformed vertex data populated by the user application or by Principia specialized fast-2D sprite components.**

D103B:Ex08 - Geometry

- **Pseudo2D vertex set geometry file. Units are in pixels. There are no normals, thus lighting should be turned off to see the object.**

```
TEXT
VS3A
TRIANGLE LIST PRIMITIVE PSEUDO2D OBJECT
@@ Vertex format specification
POS:3F, COLOR:COLOR
@@ Primitive type
4
@@ Indexed flag
0
@@ Reserved bootstrap section
0
@@ Local material code
0
@@ Total number of vertices and indices
6
0
@@ Number of frames
1
@@ Frame#0: Primitive first vertex index, primitive count, number of vertices
0  2  6
@@ Frame#0: Locus of first frame in index array and index count per frame
0  0
@@ Vertex data Trig01
000.0   000.0   0.000  FFFF0000
320.0   000.0   0.000  FF00FF00
000.0   320.0   0.000  FF00FF00
@@ Vertex data Trig01
000.0   320.0   0.000  FF00FF00
320.0   000.0   0.000  FF00FF00
320.0   320.0   0.000  FF0000FF
```

D103B:Ex08 - Geometry

- **Implementation: Camera settings for simulated 2D render. With this camera, pixel-sized vertex coordinates map to screen-centered pixels.**

```
##define (CAMERA_ST2D) as <CAMERA_3A>
   #tag Target           =  (               NONE )  ' Render on the back buffer
   #tag Viewport         =  (  - ,  - ,   - ,   - )  ‘ Use full extent of render target
   #tag ViewSim2D        =  (                 1 )
   #tag MaterialBgn      =  (          M_STDNULL )
   #tag MaterialEnd      =  (          M_STDNULL )
```

- **A pixel-scale kinex positions the geometry relative to the screen center:**

```
##define (G_EX07) as <VERTEXSET_3A>
   #tag File      =  "Files_Media\Mesh_Pseudo2D_Rectangle.vsf"
   #tag Definer   =  ( VTD_FVF_FILE )
   #tag LoadNow   =  (           1 )

##define (R_EX07) as <MESH_3A>
   #tag Instance  =  (     INST_REF )
   #tag Component =  (           0 ,    M_BASEFVF ,    G_EX07 )

##define (O_EX07) as <GENOBJECT_3A>
   #tag Instance  =  (     INST_REF )
   #tag Construct =  (  "Decl(M); Mov(-128.0,-128.0,0.0)" ,   R_EX07 ,       NONE )
```


Chapter 1:03
Basic Rendering
Principia v3.0
Reference Series
(c) Western Star
Entertainment Ltd, 2003-2006

D103C – Materials

- Materials define the appearance of the rendered geometry. They instruct the GPU how to shade rasterized primitives.
- Once a material is set, its individual properties (shaders, textures…) remain in force for all primitives rendered.
- Principia materials encapsulate many different data items used to define the rendering process. Chapter 3 covers the basics of defining and using materials.

D103C – Materials

- **Principia materials contain:**
  - GPU states (S)
  - Textures (S)
  - Shaders (S)
  - Shader data bindings (S)
  - Effects (A)
  - Reference to other embedded materials (A)
  - Phase (S)
  - Conditional logic (A)
- **Material elements are of two types:**
  - States (denoted by S above) set a GPU operation property that remains in force until removed by another material.
  - Actions (denoted by A above) operate on graphic data related to the material at the time of its GPU execution.

D103C – Materials

- **To document new features**
  - Phases
  - Keyframes
  - Gpu data settings

D103C – Materials

GPU states:

- Render pipeline states (several states matter only with the legacy fixed function pipeline, but many do not – such as depth or frame buffer settings).
- Texture sampling states (per stage).
- Texture stage states (legacy fixed pipeline, not relevant when using shaders).
- GPU material surface properties (legacy fixed pipeline, not relevant when using shaders).
- If the current trends continue, state-based GPU control will increasingly be replaced by programmable shaders.

D103C – Materials

GPU textures:

- Bind graphic components (textures, cube maps) to the GPU texture registers. Usually, the appearance of an object is determined by a variety of texture maps used to render it.

GPU shaders:

- Vertex shaders
- Pixel shaders
- Texture shaders (not very useful, but supported)
- Microsoft DX HLSL effects (not very useful but supported)
- Tesselation shaders (future capability on DX10)
- Geometry shaders (future capability on DX10)
- Buffer/fragment shaders (future capability on DX11+?)

D103C – Materials

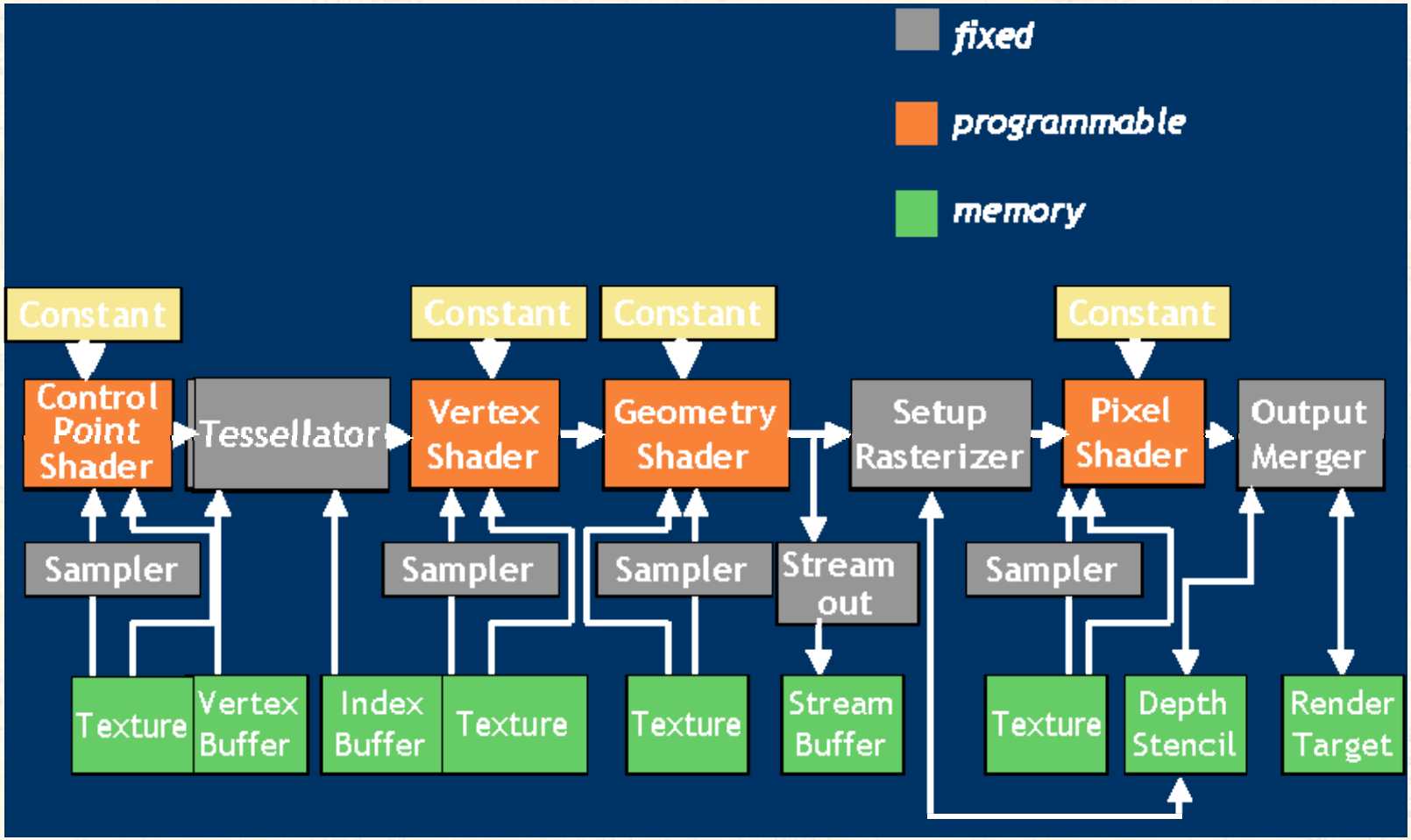
- **GPU shader data bindings:**
  - Connects application data items (transform matrices, variables) to shader program input registers
- **Principia effects:**
  - Perform pre-defined operations on graphic data or GPU settings when the material is presented to the GPU (e.g. render target selection, texture data modification...etc.)
- **Principia phase:**
  - Sets the Principia graphic interface phase. The phase determines how the interface will process data. Used for advanced SFX and performance-oriented design.
- **Material logic conditions:**
  - Controls what pieces of the material definition are actually applied based on variables describing platform capability.

D103C – Materials

- **Understand how graphic data is rendered !**
  - With current graphic hardware, you cannot simply say “this item is made of sparkling crystal with a blue halo around it”.
  - You need to understand and design how geometry data and GPU settings will render up to the desired image/style. You also need to understand the connections between geometry, materials and application/Principia data items.
  - You need to translate this design into a rendering sequence and materials applied at the proper sequence stages.
- **Chapter 3 is the starting point for building this skill. If you have attempted your own graphics programming, you will appreciate how much Principia simplifies the task of rendering.**

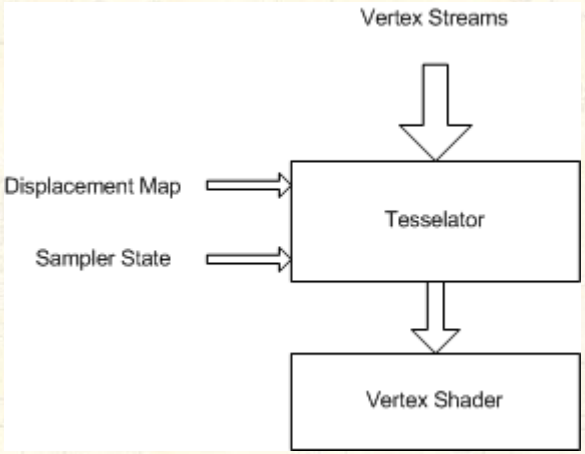
D103C – Materials

- Modern GPU rendering pipeline and core data

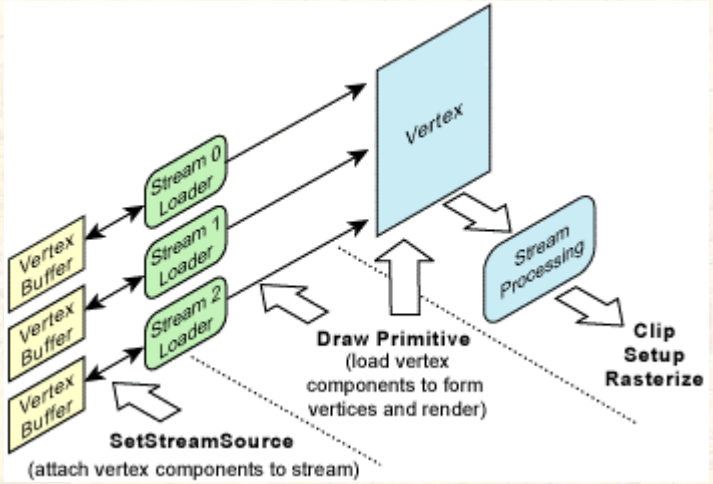


D103C – Materials

- **Front-end optional tesselation stage can take rough geometry and "enrich" it with texture encoded data. Covered in later chapters.**

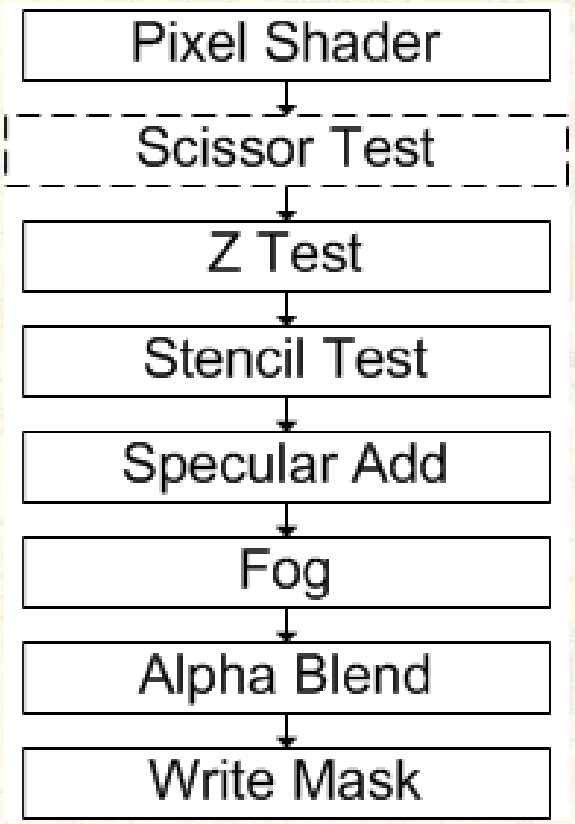


- **Binding geometry data. GPU states do not change once streams begin rendering. Geometries that appear to be made of multiple materials rendered in pieces, or are carefully designed illusions.**



D103C – Materials

- **Post-pixel rendering pipeline stages. Vital for proper generation of combined images and many SFX.**
- **Future GPUs may operate on different pipeline designs – e.g. post-DX10 designs may feature coverage shaders to replace current post-pixel processing.**



D103C – Materials

How Principia and you make use of materials?

- Materials are defined in script (occasionally, Principia will generate materials internally, but this is done for specialized circumstances).
- Graphic data that is part of the material (e.g. texture maps) is produced externally and defined in script. Principia may complement graphic data (e.g. fill instance buffers or transformed vertex buffers) using specialized components configured in script.
- The rendering sequence is constructed in script (remember the viewer-world-layer-object-kinex-mesh (material+geometry) thing?). This is where materials are specified.
- As Principia runs the rendering sequence, it schedules materials while building the rendering queue for the current scene. This is referred as the material being executed by the API. When the GPU takes over and executes the queue, the material is actually executed by the GPU as part of the physical rendering process.

D103C – Materials and Maps

Maps:

- Maps are textures used to encode data used to render the surface appearance of the current primitive.
- Maps are vital ingredients of materials. As textures, they are simply bound to the appropriate texture stage indexes.

Map generation:

- External art pipeline
- Principia procedural pipeline
- Output of render operations

Multiple render passes and compositing:

- To generate the final image, the same geometry (or variation thereof) is rendered multiple times (often on different render targets, some of which may serve as maps) with different material settings. This is a common practice for many VFX.

D103C – Materials and Maps

Commonly used map types:

- *Base color maps*: provide fine-scale color surface detail. This is the classic, ordinary use of textures...
- *Light maps*: modulate the illumination on the current primitive. Used to encode the full, complex play of light in the scene to aid or replace shaders. *Shadow maps* are special dynamically generated light maps.
- *Environment/image maps*: specialized light maps used to represent reflection or refraction. Image maps can replace 3D render portions.
- *Normal maps*: used to simulate fine-scale surface detail and its interaction with light.
- *Displacement maps*: similar to normal maps, but used to actually move the geometry rendered.
- *Specular and other property maps*: used to represent fine-scale variation of light interaction properties.
- *Depth/occlusion maps*: encode distance information and other geometric relationships used by advanced shaders.
- *Function maps*: used to represent functional relationships between inputs and outputs, such as those involved in shaders.

D103C – Materials and Textures

- **Principia textures and materials:**
  - Textures are an essential ingredient of materials, as maps and more…
  - Principia encapsulates textures in the <SURFACE> script component and GX_Surface programmatic API component.
  - The <SURFACE> component is a rich encapsulation of data that supports the myriad of uses to which textures are put.
  - It features multiple properties and tags, which are enumerated in D103I, and introduced throughout the demo materials.

D103C – GPU States Synopsis

```
D3DRS_ZENABLE = 7,
D3DRS_FILLMODE = 8,
D3DRS_SHADEMODE = 9,
D3DRS_ZWRITEENABLE = 14,
D3DRS_ALPHATESTENABLE = 15,
D3DRS_LASTPIXEL = 16,
D3DRS_SRCBLEND = 19,
D3DRS_DESTBLEND = 20,
D3DRS_CULLMODE = 22,
D3DRS_ZFUNC = 23,
D3DRS_ALPHAREF = 24,
D3DRS_ALPHAFUNC = 25,
D3DRS_DITHERENABLE = 26,
D3DRS_ALPHABLENDENABLE = 27,
D3DRS_FOGENABLE = 28,
D3DRS_SPECULARENABLE = 29,
D3DRS_FOGCOLOR = 34,
D3DRS_FOGTABLEMODE = 35,
D3DRS_FOGSTART = 36,
D3DRS_FOGEND = 37,
D3DRS_FOGDENSITY = 38,
D3DRS_RANGEFOGENABLE = 48,
D3DRS_STENCILENABLE = 52,
D3DRS_STENCILFAIL = 53,
D3DRS_STENCILZFAIL = 54,
D3DRS_STENCILPASS = 55,
D3DRS_STENCILFUNC = 56,
D3DRS_STENCILREF = 57,
D3DRS_STENCILMASK = 58,
D3DRS_TEXTUREFACTOR = 60,
D3DRS_WRAP0 = 128,
. . .
D3DRS_WRAP15 = 205,
D3DRS_SEPARATEALPHABLENDENABLE = 206,
D3DRS_SRCBLENDALPHA = 207,
D3DRS_DESTBLENDALPHA = 208,
D3DRS_BLENDOPALPHA = 209,
D3DRS_MULTISAMPLEANTIALIAS = 161,
D3DRS_MULTISAMPLEMASK = 162,
D3DRS_CLIPPING = 136,
D3DRS_LIGHTING = 137,
D3DRS_AMBIENT = 139,
D3DRS_FOGVERTEXMODE = 140,
D3DRS_COLORVERTEX = 141,
D3DRS_LOCALVIEWER = 142,
D3DRS_NORMALIZENORMALS = 143,
D3DRS_DIFFUSEMATERIALSOURCE = 145,
D3DRS_SPECULARMATERIALSOURCE = 146,
D3DRS_AMBIENTMATERIALSOURCE = 147,
D3DRS_EMISSIVEMATERIALSOURCE = 148,
D3DRS_VERTEXBLEND = 151,
D3DRS_CLIPPLANEENABLE = 152,
D3DRS_POINTSIZE = 154,
D3DRS_POINTSIZE_MIN = 155,
D3DRS_POINTSPRITEENABLE = 156,
D3DRS_POINTSCALEENABLE = 157,
D3DRS_POINTSCALE_A = 158,
. . .
```

D103C – GPU States Synopsis

D3DRS_PATCHEDGESTYLE = 163,
D3DRS_DEBUGMONITORTOKEN = 165,
D3DRS_POINTSIZE_MAX = 166,
D3DRS_INDEXEDVERTEXBLENDENABLE = 167,
D3DRS_COLORWRITEENABLE = 168,
D3DRS_TWEENFACTOR = 170,
D3DRS_BLENDOP = 171,
D3DRS_POSITIONDEGREE = 172,
D3DRS_NORMALDEGREE = 173,
D3DRS_SCISSORTESTENABLE = 174,
D3DRS_SLOPESCALEDEPTHBIAS = 175,
D3DRS_ANTIALIASEDLINEENABLE = 176,
D3DRS_MINTESSELLATIONLEVEL = 178,
D3DRS_MAXTESSELLATIONLEVEL = 179,
D3DRS_ADAPTIVETESS_X = 180,
. . .
D3DRS_ADAPTIVETESS_W = 183,
D3DRS_ENABLEADAPTIVETESSELATION = 184,
D3DRS_TWOSIDEDSTENCILMODE = 185,
D3DRS_CCW_STENCILFAIL = 186,
D3DRS_CCW_STENCILZFAIL = 187,
D3DRS_CCW_STENCILPASS = 188,
D3DRS_CCW_STENCILFUNC = 189,
D3DRS_COLORWRITEENABLE1 = 190,
D3DRS_COLORWRITEENABLE2 = 191,
D3DRS_COLORWRITEENABLE3 = 192,
D3DRS_BLENDFACTOR = 193,
D3DRS_SRGBWRITEENABLE = 194,
D3DRS_DEPTHBIAS = 195

Per-stage sampler states:

D3DSAMP_ADDRESSU = 1,
D3DSAMP_ADDRESSV = 2,
D3DSAMP_ADDRESSW = 3,
D3DSAMP_BORDERCOLOR = 4,
D3DSAMP_MAGFILTER = 5,
D3DSAMP_MINFILTER = 6,
D3DSAMP_MIPFILTER = 7,
D3DSAMP_MIPMAPLODBIAS = 8,
D3DSAMP_MAXMIPLEVEL = 9,
D3DSAMP_MAXANISOTROPY = 10,
D3DSAMP_SRGBTEXTURE = 11,
D3DSAMP_ELEMENTINDEX = 12,
D3DSAMP_DMAPOFFSET = 13,

This list is meant to serve as a quick recall help. Principia provides named definitions of all states and their values. The names are based on the DirectX names, and listed in the Principia Reference Manual. It is a good idea to become conversant in the meaning and usage of GPU states.


Principia V3.0.0 Demos
Ex01: Reading texture material data from a 3DS file.
Textures=2, Mesh=5, Matl=2, Shaders=YES
FPS: 66.7
GPS: 31.3
Lighting on
Specular on
Perspective
Orthogonal
View
Lite1
Lite2
Lite3
Matl
Camera Distance
6.0
Camera Elevation
25.0
Camera Azimuth
-58.0
Camera FoV
25.0
At Point
0.0
0.0
This simply textured object is directly loaded from a 3DS file. The Principia 3DS loader will automatically load material textures and fixed function pipeline properties, and properly assign them to their vertex streams. This is one of the few cases where the user does not need to focus much on materials … it is also not a very good practice.

D103C:Ex01 – Materials

Objectives

- Load and render a textured geometry from a 3DS file
- Show the use of base color maps (ordinary textures)
- Show the use of UID referencing for external components

Requirements and Implementation

- Configure the 3DS loader to import an object with multiple texture-bearing materials in a manner that preserves texturing at the seams.
- Script the textures referenced by the 3DS materials, and assign them an UID matching texture names in the 3DS file.
- Construct the renderable object and render it using the standard Blinn/Gouraud shaders from SHD3D.

Notes

- When rendering with shaders, any substrate material properties (except Texture0) directly loaded from 3DS will be ignored, unless explicitly passed to shader registers.

D103C:Ex01 – Materials

IDs, UIDs and referencing:

- Referencing: Most Principia components need to refer to other components as part of their make-up. Principia uses two mechanisms for that: IDs and UIDs.
- IDs are unique numbers that correspond to the names used when an object is created in script. For instance, ##define (TEX) as <SURFACE> creates a GX_Surface component with an ID of say 90565, which is mapped to the string “TEX” internally. When Principia sees TEX in script, it automatically translates that to 90565 using its internal algorithms, and knows which object one is referring to. In fact, all the names you see in script are translated to numbers for speed.
- IDs are local to their scripts. In two different scripts, “TEX” may map to 90565 or 93454 or anything else. This presents a problem when loading a component from a storage file instead of a script. The file may be loaded by many different scripts, yet we want all references contained in the file to be accurate. Moreover, the script writer may want to call the same component something else but “TEX” in another application.
- Principia uses UIDs when storing or loading references from external files (not scripts). The UIDs are text strings that are designed to maximize the portability of save files to be used by Principia. All AX_Grafic() and many other classes of Principia components have UIDs.

D103C:Ex01 – Materials

UIDs specification rules:

- Explicit <#tag UID> forces a name on the component. Regardless of ID, when referenced in files by this UID, the component will always be found.
- If no tag is present, implicit condensation of the FName internal variable into an UID. This is often used with textures, and enables us to locate textures referenced by materials by file name.
- If no FName is present, the string equivalent of the ID will be used to construct an UID. This means that script names of the same component across different scripts must be consistent.
- File storage of UIDs. When components are stored in a file using the Principia ::File*() methods, the current UID will be stored. When loading components from the file, their UID will be restored, unless overridden by the <UID> tag.

Example:

e mesh "MS_HEAD" which contains the material "SkinSetUp.mat" which contains the texture "SubsurfaceMap.png". All these components need to be either scripted with corresponding UIDs, or loaded from an external file orehand, again with the same UIDs.