

NPSGDL: An Object Oriented Graphics Description Language for Virtual World Application Support

Kalin P. Wilson, Michael J. Zyda* and David R. Pratt

Naval Postgraduate School

Department of Computer Science

Monterey, California 93943-5100 USA

*contact author. Email: zyda@cs.nps.navy.mil

Abstract

Many virtual world applications today represent the cutting edge in real-time 3D interactive graphics. These applications must model many complex, often changing, graphical objects. These graphical objects must be modeled both visually and behaviorly. An application independent method for describing graphical objects is essential for rapid prototyping and development. This paper presents a simple, flexible and extensible graphics description language, NPSGDL, used in virtual world development at the Naval Postgraduate School.

Keywords: Virtual Worlds, Object-Oriented, Graphics, Description Language

Introduction

The graphical objects used in today's virtual world applications are usually complex. The typical object may consist of several hundred polygons, several lighting materials and textures. Also objects may be associated together into more complex objects which support articulated and/or animated features. The need for an application independent way to describe, manipulate and manage any variety of graphical objects was recognized during research on early systems in the Graphics and Video Laboratory of the Department of Computer Science at the Naval Postgraduate School [1,2]. NPSGDL is the latest system developed to address this need.

NPSGDL provides two levels of application support to the developer. At the highest level, NPSGDL is a language system that can be used to describe and manage complete graphical objects

as individual entities. At the lower level, NPSGDL is a collection of classes representing graphics primitives. These classes can be used directly by the developer independent of the language system. Since NPSGDL is designed and implemented in the object oriented paradigm, each of the classes that define the language system can be extended using inheritance to provide more specific application support.

NPSGDL is implemented in C++ for use on Silicon Graphics Inc. IRIS workstations. This paper describes the design, implementation and use of NPSGDL.

Previous Work

Graphics developers and users naturally think of graphics entities as *objects*. Thus, the object oriented paradigm is natural for designing and implementing graphics applications. Wisskirchen [3,4] has examined applying the object-oriented paradigm to existing graphics standards with success. Egbert, et al [5] has developed a system that supports graphics applications at several levels. Their system abstracts the graphics primitives and rendering process to a level that is more portable and accessible to the applications developer. Other systems provide more specific support for such things as graphics animation and physical modeling [6].

Previous systems developed at the Naval Postgraduate School have concentrated on graphical object storage and retrieval [2]. These began as binary and ASCII file formats and evolved into more complex systems.

NPSGDL incorporates many of the concepts and capabilities of the above systems. In particular, NPSGDL provides graphics primitives and

higher level application support (in the form of simple animation and viewpoint control). In contrast to some of the systems mentioned above, NPSGDL is more tightly coupled to the rendering software and hardware used in the Graphics and Video Laboratory. This helps facilitate the efficiency needed in our virtual world applications. In addition, NPSGDL incorporates a language system that provides many advantages:

- Off-line object definition and modification.
- Object storage and retrieval.
- Object sharing between applications.
- Application independence of objects.

NPSGDL gives application developers a high level of abstraction that supports rapid prototyping and development of complex graphics systems. Application developers can concentrate on how objects behave and are used rather than the rendering of the object. The object-oriented nature of the system provides for easy extension and modification. The language system facilitates the creation and maintenance of collections of graphical objects that can be used and shared by many researchers. It also serves to standardize the representation of graphical objects across applications.

Design

The design of NPSGDL seeks to satisfy many diverse goals:

1. Backward compatibility with previous NPSOFF [2] definition files.
2. Easy extension and addition of language elements.
3. Simple maintenance.
4. Simple, easy-to-understand user interface.
5. High efficiency with respect to graphics intensive applications.

These design goals were addressed using the object-oriented paradigm and C++. The object oriented paradigm was the ideal choice for many reasons. Foremost is the paradigm's support for a naturally expressive design and the close correspondence between the design and implementation. The abstraction, encapsulation and inheritance facilities inherent in the object oriented para-

digm directly supported goals 2 and 3.

The C++ language was chosen for:

1. Its compatibility with existing C libraries.
2. Its efficiency (related to C compatibility).
3. The degree of control and flexibility afforded the designer/implementor over object and non-object oriented construct usage.
4. The availability of implementations on Silicon Graphics Inc. workstations.

The design of NPSGDL was primarily responsibility driven [7] and atomically oriented. Because NPSGDL is a library of classes available to the application developer, each leaf class is basically stand-alone in functionality. We did not use function specific classes (e.g. a displayer) because we wanted to approach the ideal of adding a new class to the main hierarchy without affecting any other class modules.

The NPSGDL language is divided into atomic units called *Tokens*. Each Token is responsible for:

1. Reading itself from an ASCII stream.
2. Writing itself to an ASCII stream.
3. Defining itself if applicable (explained below).
4. Displaying itself if applicable.

In addition, a Token can be copied and compared for equality and order.

The set of Tokens is divided into five major subsets: Deftokens, Settokens, Drawtokens, Xformtokens and Othertokens. Each of these is described in the following section.

The Token hierarchy provides the major functionality for the NPSGDL language system. In order to support this functionality, several underlying systems had to be built. These systems are a run-time type identification system, an object persistence system and a simple garbage collection system. Most of these systems are provided by other language environments. However these systems and the functionality they support are an inherent part of the design. Language support for systems such as these is an important consideration and can affect design decisions.

The other hierarchy in NPSGDL is the *NPSobject* hierarchy. An NPSobject is the user's normal

interface to the language system. An NPSObject's basic responsibilities are:

1. Read/ initialize from an Ascii stream.
2. Write to an Ascii stream.
3. Define itself.
4. Display itself

At the high level, an NPSObject represents a graphical object used in an application. These can be as simple as a single polygon or sphere or as complex as an aircraft or ground vehicle. Also NPSObjects can be used to represent objects that do not display such as an object that contains all of the lighting material definitions for an application. At the low level, an NPSObject is a collection of tokens that represents the properties and depiction of the NPSObject. In normal usage, an NP-Object is semantically and logically associated with an Ascii file containing token definitions. This file represents the definition of the NPSObject. Although this is the normal usage of the NPSGDL system, the user is not limited to this mode. The individual Token subclasses can be used directly and initialized at run-time. Likewise, NP-Objects can be created and modified at run-time independent of a definition file. This provides a flexible framework for both application developers using NPSGDL and tool developers manipulating NPSObjects and Tokens.

Animation and Multi-Resolution Display

Two of the important capabilities that NPSGDL supports are simple animation and multi-resolution display. Animation is supported through user defined variables that are automatically updated by the NPSGDL system. These variables are similar to the animated basic types described in [9]. The user can place simple constraints on the variable or specify a user defined rule that the variable uses to update itself. An animated variable can be used by a variety of tokens in place of fixed values. The animated variable (Defvariable) acts as the server and other tokens as clients. The client token sets up an association with the variable and informs the variable to update itself prior to each display cycle. This scheme

supports a wide range of simple, continuous animations. For example, if a developer wanted to model a lighthouse with a rotating beacon, the rotation could be controlled by an animated variable using constraints defined by the developer. The beacon rotation would then be updated entirely by NPSGDL leaving the developer to concentrate on higher level details. If more complex behavior is needed by the variable, the developer can specify a rule (function) for updating the variable. The name of the rule is given and the variable associates to the rule at run time from a table initialized by the user. This rule is then used to update the variable while the variable maintains the user constraints. Animated variables are discussed in more detail below. See Figure 1 below for an example using animated variables.

Multi-resolution display is important to complex virtual world applications. Our experience has been that most of our applications are graphics bound. Much of our effort goes to reducing the number of polygons sent to the renderer. NPSGDL supports three drawing resolutions for each of its Drawtokens: high, medium and low. NPSGDL supports both a simple and a hierarchical resolution scheme. An NPSObject can be displayed at any of the three resolutions under user control. The Drawtokens are displayed in either a hierarchical or single level mode. Single level mode displays only the tokens having a particular resolution. In hierarchical display mode, high resolution displays all Drawtokens, medium resolution displays medium and low resolution Drawtokens and low resolution displays only low resolution tokens. Thus an object designer can designate which polygons, subobjects etc. are visible at each resolution. The application developer can then display NPSObjects at varying resolutions; based on distance from the viewer for instance. This multi-resolution model supports two methods of object design. The designer can specify three distinct representations of an object or the designer can design using one set of surfaces but designate which are visible at various resolutions. The latter is much more difficult but results in smaller object sizes. Using this multi-resolution scheme, we have seen substantial performance improvements,

as is expected. The primary cost is development time of the object models.

Language Description

The NPSGDL language contains approximately forty-one tokens. Each token belongs to one of five groups: Deftokens, Settokens, Drawtokens, Xformtokens and Othertokens. Each token is represented by a separate class with the token groups serving as super-classes. The NPSGDL language is context-sensitive with the parsing responsibility distributed across the token hierarchy. Each language token consists of an identifier followed by a known or limited number of subcomponents representing properties of the token and associated values. Token identifiers and subcomponent identifiers are case insensitive and the order of subcomponents is generally not important. This provides a simple flexible syntax that is easy to remember. The user need only remember property names and valid values, not necessarily the order that properties must be listed. The identifier for each token in the language is known to the root of the hierarchy and accessible by a class method that reads a stream, recognizes identifiers and creates specific token instances, which initialize themselves with the subcomponents. When a token instance has initialized itself, the stream is positioned at the next identifier and the process continues.

The following sections describe the specific token groups. Simple examples are given for each of the tokens illustrating their formats. All values are floating point. Color values are given in the range 0.0 - 1.0. For the sake of brevity, color values are denoted by "r g b" and optionally "a" for alpha values. Also the term "vertex" is used to represent three component coordinate values. It is important to note that many of the components of a token offer reasonable defaults and are not required.

Deftokens

Deftokens represent graphical entities that are stored for later access and use. Some Deftokens correspond to system settings that must be defined prior to use. Others simply store information to be

used later. A good example of a Deftoken is a lighting material definition which must be system defined before it can be used for rendering. Each Deftoken sub-class provides a random access table which stores copies of each instance created. These tables are available to complimentary Settokens and the user. Thus each Deftoken facilitates pools of instances that can be accessed as needed. Each Deftoken has a string name that uniquely identifies it. The Deftoken class is responsible for reading and writing this name. Individual Deftoken sub-classes are responsible for any subcomponents after the name. All Deftoken formats begin with the token identifier and end with "defend". This allows the property identifiers and values to be given in any order as discussed above. The current Deftoken sub-classes are:

Defmaterial: This token represents a lighting material definition. Material definitions specify how light interacts with a surface. These definitions are made known to the graphics system and are available to the rendering process as needed. The Defmaterial token has the following format:

```
defmaterial name
  ambient r g b
  diffuse r g b
  emission r g b
  specular r g b
  shininess f
  alpha f
defend
```

Deflight: This token represents a light definition. Light definitions are used during rendering to color and shade visible surfaces. Each light is pre-defined to the graphics system for quick access during rendering. Deflights support local and infinite light sources, colored lights and spotlights as specified in [8]. The deflight format is:

```
deflight name
  ambient r g b
  lcolor r g b
  position x y z w // w=0 infinite
                      // w=1 local
  spotlight exp spreadangle
  spotdirection x y z
defend
```

Deflmodel: This token represents a lighting model definition. The lighting model controls how lights and materials are used and the calculations employed. Example properties of a Deflmodel are: ambient light color, whether the viewer is local, attenuation factors and whether two-sided material lighting is enabled.

```
deflmodel name
  ambient r g b
  localviewer yes|no
  attenuation K0 K1 K2
  twoside yes|no
defend
```

Defcolor: This token represents a simple rgb color used for rendering when lighting is not desired.

```
defcolor name
  r g b
```

Deftexture: This token represents a texture mapping definition. It associates an image file name with the texture mapping options available from the Silicon Graphics Graphics Library. The Deftexture has the following format:

```
deftexture name
  imagefile filename
  components num
  minfilter point|biliner|
    mipmap_point|mipmap_linear|
    mipmap_bilinear
  magfilter point|biliner
  wrap repeat|clamp
  wrap_s repeat|clamp
  wrap_t repeat|clamp
  tile f f f f
  // wrap, wrap_s & wrap_t, and tile
  // are mutually exclusive
defend
```

Deftexenv: This token represents environmental settings for texture mapping. It supports texture decaling, modulation and blending as specified in [8].

```
deftexenv name
  blend|decal|modulate
  color r g b a
defend
```

Deftexgenalg: This token allows the user to

specify parameters for the automatic generation of texture coordinates for polygons. It uses the coordinate generation capabilities available through the Silicon Graphics GL library [8].

```
deftexgenalg name
  sdir linear|contour|spheremap a b c d
  tdir linear|contour|spheremap a b c d
  // a b c d specify plane eq.
  // coefficients
defend
```

Defobject: This token allows the user to group any set of tokens into a named object that is stored for use by any other object in the system. Defobjects are normally subobjects used by NPSobjects or other Defobjects. An example would be the wheel of a vehicle. One Defobject would represent the wheel. Any object needing the wheel could reference it using the Callobject token discussed below.

```
defobject wheel
  setmaterial wheelcolor
  defpoly high
    0.0 0.0 1.0
    8
    0.0 0.0 0.0
    :
    :
    -0.1 0.1 0.0
defend
```

Readobject: This token gives the user the ability to access other NPSGDL object files from an object file. The name of a Readobject token is interpreted as a filename. The file is opened and read as an NPSobject. It is assumed that the file contains Defobject definitions which are in turn stored in the Defobject table for later use.

```
readobject wheel.gdl
```

Defvariable: This token represents a user defined constrained floating point variable that is updated and maintained by NPSGDL. As explained in the animation section above, Defvariables can be used in place of values in certain tokens. In this way, Defvariables offer simple animation support within NPSGDL. The value of a Defvariable can be updated each display loop or per unit time (sec). It can oscillate between its min and max value or wrap around. Also an update

function (rule) can be named to correspond to a user provided function. This makes complex behaviors possible. The format for the Defvariable is:

```
Defvariable name
  min minval
  max maxval
  init initialval
  inc increment // per sec if timed
  timed yes|no
  wrap yes|no
  rulename rname
defend
```

Defpcamera: This token allows the user to specify a perspective viewing frustum. The Defpcamera can then be used in conjunction with a Defviewpoint to define the view volume used for rendering. The field of view for a Defpcamera can be specified by a fixed value or a Defvariable. Thus the field of view can be animated to support effects like zooming.

```
Defpcamera name
  nearplane nz
  farplane fz
  aspect aratio
  fov angle
defend
```

Defocamera: This token is similar to the Defpcamera. It supports orthographic projections. The user specifies the view volume by setting the clipping planes in each dimension.

```
Defocamera name
  xdim xmin xmax
  ydim ymin ymax
  zdim zmin zmax
defend
```

Defviewpoint: This token is used to specify the orientation of a camera. The parameters are the location of the viewpoint, the location of the reference point and the twist angle of the view volume. Each location can be specified using a combination of fixed values and Defvariable names as well as a single Defobject name. The twist angle can also be specified by a Defvariable name for continuous update. The flexible format for specifying locations and the animation support through Defvariables make Defviewpoints interesting and

easy to use.

```
Defviewpoint name
  from x y z|Defvariable name|
  Defobject name
  to x y z|Defvariable name|
  Defobject name
  twist angle|Defvariable name
defend
```

Settokens

Settokens represent entities that change the graphics state. They are normally paired with a Deftoken and provide for the use of the Deftoken's properties. Thus the Deftoken and Settoken classes are cooperative with the Deftokens containing a pool of definitions and the Settokens providing access to the definitions. Most Settokens have the property that only one (of each type) is active at any time. Each Settoken has a name. The Settoken class is responsible for reading and writing this name. Also a Settoken is usually specified as on or off. Most Settoken formats use only a single line of text so there is no end marker as with Deftokens. The different Settokens are described below.

Setmaterial: This token enables the use of a material definition. It makes the named material current for use during rendering assuming the name corresponds to a valid Defmaterial. The format for a Setmaterial is:

```
setmaterial name
```

Setbackmaterial: This token is the same as the Setmaterial token except it specifies which material to use for the backs of polygons when two-sided lighting is enabled.

```
setbackmaterial name
```

Setlight: This token allows the use of lighting definitions. A light may be on or off. Up to eight lights may be on at one time. The format for Setlight is:

```
setlight name num on|off
```

Setlmodel: This token activates a named lighting model. Setlmodels can be on or off.

```
setlmodel name on|off
```

SetColor: This token sets the current drawing color to a previously named and defined color.

```
setcolor name
```

Settexture: This token enables texturing using a previous texture definition. A Settexture can be turned on or off to allow texturing of portions of an object.

```
settexture name on|off
```

Settexenv: This token selects a texturing environment definition for use.

```
settexenv name on|off
```

Settexgenalg: This token selects the algorithm definition to use for automatic texture coordinate generation. Also the user can specify whether coordinates should be generated along the s axis or t axis or both.

```
settexgenalg name sdir on|off tdir  
on|off
```

Setpcamera: This token selects a Defpcamera definition for use. The Setpcamera can be on or off. This enables interactive control of cameras and the switching of view volume specifications between user code and NPSGDL tokens.

```
setpcamera name on|off
```

Setocamera: This token is the same as the Setpcamera but for orthographic cameras.

```
setocamera name on|off
```

Setviewpoint: This token allows the user to select a previously defined viewpoint for rendering. The Setviewpoint token is used in conjunction with the Setpcamera and Setocamera tokens to define the viewing volume location and orientation.

```
setviewpoint name on|off
```

Drawtokens

Drawtokens represent the visible graphical entities. These are the items in a scene that all other tokens affect. Most of the Drawtokens represent low-level graphical constructs but several offer higher level support. All Drawtokens have a resolution value. The Drawtoken class is responsible

for reading and writing the resolution. The resolution may be high, medium or low. The Drawtoken resolutions are used to support multi-resolution objects in applications. A description of the Drawtokens in NPSGDL follows. Here *res* denotes high|medium|low resolution choices as discussed above. Also *vertex* denotes an xyz coordinate.

Defpoly: This token represents a basic polygon. The polygon has a single normal and may have texture coordinates optionally specified. The format for a Defpoly is:

```
defpoly res  
nx ny nz // normal  
numvertices  
vertex1 [s1 t1]  
:  
vertexn [sn tn]
```

A_defpoly: This token is the same as the Defpoly except that it supports the use of Defvariable names in place of vertex coordinate values. The “A_” prefix denotes animated behavior. The ability to specify polygon vertex coordinates that change automatically is powerful and fun.

```
a_defpoly res  
nx ny nz  
numvertices  
vertex1|defvariable name [s1 t1]  
:  
:  
vertexn|defvariable name [sn tn]
```

Defsurface: This token represents a polygonal surface with vertex normals. Also texture coordinates can be optionally specified for each vertex. The format for a Defsurface is:

```
defsurface res  
numvertices  
vertex1 normal1 [s1 t1]  
:  
vertexn normaln [sn tn]
```

Defmesh: This token represents a polygonal surface using a triangular mesh. It uses the same basic format as the Defsurface.

```
defmesh res  
numvertices  
vertex1 normal1 [s1 t1]
```

```

:
vertexn normaln [sn tn]

```

Defqstrip: This token represents a polygonal surface using quadrilateral strips. It uses the same basic format as a Defsurface.

```

defqstrip res
  numvertices
  vertex1 normal1 [s1 t1]
:
  vertexn normaln [sn tn]

```

Callobject: This token allows the user to access and display Defobjects. A Callobject is used to display subobjects that are defined separately. A Callobject token has two resolutions associated with it. One for itself pertaining to which resolutions the subobject will be shown and another to designate the display resolution for the subobject. The format for a Callobject is:

```

callobject res objname display-res

```

Defcircle: This token represents a simple 2D circle defined in the x-y plane. The properties of a Defcircle are its center location and radius.

```

defcircle res
  center x y z
  radius r
defend

```

Defsphere: This token allows the user to define a sphere parametrically. In addition to center and radius, the user can specify the number of polygons to use to represent the sphere. Its format is:

```

defsphere res
  center x y z
  radius r
  panels numpanels
endsphere

```

Defcylinder: This token allows the user to parametrically specify a cylinder for display. Its format is similar to Defsphere with the addition of a height property.

```

defcylinder res
  center x y z

```

```

radius r
height h
panels numpanels
defend

```

Defcone: This token allows the user to parametrically define a cone for display. Its format is similar to the Defcylinder format.

```

defcone res
  center x y z
  radius r
  height h
  panels numpanels
defend

```

Defline: This token allows the user to specify a multi-point line in three dimensions.

```

defline res
  numpoints
  vertex1
:
:
  vertexn

```

Defdecal: This token provides the user with a facility to define decaled polygons. Decaling is a technique for properly rendering co-planar polygons while using z-buffered hidden surface elimination. With the Defdecal token users can specify the components of the underlay and overlay portions of a decal. The components of the underlay and overlay can be any displayable token (Drawtoken, Settoken and Xformtoken). In normal use only Setmaterial and Defpoly tokens are used.

```

defdecal res
  underlay
  defpoly1 // see format above
  defpoly2
:
  overlay
  defpoly1
  defpoly2
:
defend

```

Xformtokens

Xformtokens represent entities that alter the normal representation of Drawtokens. The normal transformations of rotation, translation and scaling are represented. Also there are tokens that re-

late directly to capabilities of the Silicon Graphics hardware and rendering process. Following is a brief description of each of the Xformtokens.

Rotate: This token represents a single axis rotation. The rotation is normally performed in world space with the rotation specified by a float representing whole and fractional angles. The format for a Rotatetok is:

```
rotate x|y|z angle
```

A_Rotate: This token is the animated version of a Rotatetok. It accepts a Defvariable name for the rotation value. This token is useful for simple animations like wheels or propellers turning. It allows the developer to specify the animation constraints and leave the rest to the NPSGDL system.

```
a_rotate x|y|z angle|defvariable name
```

Translate: This token represents a simple translation in 3-space. The format is:

```
translate dx dy dz
```

A_Translate: This token is the animated version of the Translatetok. Any of the three translation values can be linked to a Defvariable for automatic update.

```
a_translate dx|name dy|name dz|name
```

Scale: This token represents a 3D scale.

```
scale sx sy sz
```

A_Scale: This token allows the use of Defvariables for animated scaling.

```
a_scale sx|name sy|name sz|name
```

Pushmatrix: This token gives the user access to the pushmatrix function in the SGI GL [8] library. This function saves the state of the rendering transformation matrix.

```
pushmatrix
```

Popmatrix: This token gives the user access to the popmatrix function in the SGI GL [8] library. The popmatrix function removes the current transformation matrix from the matrix stack.

```
popmatrix
```

Loadmatrix: This token takes a user defined 4x4 matrix and initializes the hardware matrix

stack with it.

```
loadmatrix  
a b c d  
e f g h  
i j k l  
m n o p
```

Multmatrix: This token allows the user to multiply a user-defined 4x4 matrix onto the matrix stack.

```
multmatrix  
a b c d  
e f g h  
i j k l  
m n o p
```

Loadunit: This token allows the user to initialize the transformation matrix stack with unit matrix.

```
loadunit
```

Othertokens

Othertokens are auxiliary tokens that do not directly affect the graphical representation of an object. They are not related except for this fact. The current Othertokens are described below:

Name: This token allows the user to name an NPSobject. When a NPSGDL definition file is read by an NPSobject the object scans for Name tokens and saves the last one for user inquiries. This is mostly for documentation use.

```
name objname
```

Origin: This token specifies the origin or reference point for an NPSobject. Like the Name token, an NPSobject scans the definition file for Origin tokens. At run time, the user can query an NP-Object for its origin. This information is useful for transformations and viewing.

```
origin 0 0 0
```

Comment and Lcomment: These tokens facilitate C++ style comments in NPSGDL definition files. The Comment token is delimited by the “/*” “*/” pair and can be multi line. The Lcomment is a single line comment and is delimited by “//”.

Overall, the NPSGDL language is fairly simple and easy to remember and understand. Yet it offers a great deal of flexibility and power through

```

name planets
origin 0 0 0

/* This gdl file represents a earth-like
planet and a single moon
The two planets are textured
and are rotated using animated
variables.
*/
// define the Sun
deflight sun
ambient 0 0 0
lcolor 1 1 0.75
position -1 0 0.25 0
defend

// define some materials
defmaterial sky_blue
emission 0.0 0.0 0.0
ambient 0.105882 0.161569 0.200000
diffuse 0.529412 0.807843 1.000000
specular 0.0 0.0 0.0
shininess 0.0
alpha 1.0
defend

defmaterial aquamarine
emission 0.0 0.0 0.0
ambient 0.099608 0.200000 0.166275
diffuse 0.498039 1.000000 0.831373
specular 0.0 0.0 0.0
shininess 0.0
alpha 1.0
defend

defmaterial gray4
emission 0.0 0.0 0.0
ambient 0.007843 0.007843 0.007843
diffuse 0.039216 0.039216 0.039216
specular 0.0 0.0 0.0
shininess 0.0
alpha 1.0
defend

```

Figure 1

```

// define the textures
deftexture earth
imagefile earthclouds.rgb
minfilter mipmap_bilinear
magfilter bilinear
wrap repeat
endtexture // components default = 4

deftexture moon
imagefile moon.rgb
minfilter mipmap_bilinear
magfilter bilinear
wrap repeat
endtexture

deftexenv planetenv
modulate
defend

deftexgenalg earthalg
sdir linear 0.0 0.1 0.5 20.0 // plane equations
tdir linear -0.10 0.0 0.1 20.0
defend

deftexgenalg moonalg
sdir linear 0.0 0.1 0.1 0.0
tdir linear -0.1 0.0 0.1 0.0
defend

// the animated variables for rotating the planets
defvariable earthrot
min 0
max 360
init 0
inc .1
timed yes
wrap yes
defend

defvariable moonorbit
min 0
max 360
init 0
inc 0.15
timed yes
wrap yes
defend

```

Figure 1 (cont.)

```

defvariable moonrot
  min 0
  max 360
  init 0
  inc 0.2
  timed yes
  wrap yes
defend
// this is the displayable portion
setlight sun 2 on
settexenv planetenv on

// save the state of the application transformation
// matrix
pushmatrix

// incline the scene
rotate z 23.5

pushmatrix
// spin the earth continuously
a_rotate y earthrot

// color and texture the earth
setmaterial sky_blue
settexgenalg earthalg sdir on tdir on
setttexture earth on

// draw the earth, make it low res visible
defsphere low
  center 0 0 0
  radius 15
  panels 200
endsphere
setttexture earth off

// draw the axis
setmaterial gray4
defcylinder med
  center 0 -20 0
  radius 0.5
  panels 10
  height 40
endcylinder

popmatrix

```

Figure 1 (cont.)

```

// isolate the moons motion
pushmatrix

// continously rotate the moon about the earth
a_rotate y moonorbit
translate -40 0 0

// isolate the moons spin
pushmatrix

a_rotate y moonrot

// color and texture the moon
setmaterial aquamarine
settexgenalg moonalg sdir on tdir on
setttexture moon on

defsphere high
  center 0 0 0
  radius 5
  panels 100
endsphere
setttexture moon off

// use a predefined material for an object on the
// moon
setmaterial brass
rotate z -90
defcone medium
  center 0 0 0
  radius 1.5
  height 7
  panels 10
endcone

// recover the state of the transformation matrix
popmatrix
popmatrix
popmatrix

```

Figure 1 (cont.)

higher level support for graphical abstractions. A simple example is shown in Figure 1. In it, a simple model of the earth and moon is described. The earth and moon are textured spheres and both rotate under NPSGDL control, both about their axis

and in orbit in the case of the moon.

Implementation

Many issues came up during implementation related directly to the use of C++ as the implementation language. Because of the lack of a language standard, no built-in support for run-time type information, no garbage collection and the lack of standard data structure classes, a great deal of time was spent developing these basic systems and structures before implementing NPSGDL.

The first effort was to construct a library of standard data structure abstract data types and concrete data types. Initially several popular public domain libraries were considered including the National Institutes of Health Class Library (NIHCL)[10] and the Texas Instruments, Inc. Library (COOL)[11]. Although the implementation of both libraries was very educational, neither of these libraries was used for several reasons:

1. The NIHCL single root hierarchy was deemed inappropriate.
2. Designing classes to be used by the libraries was cumbersome.
3. The time investment to become proficient using the libraries was too high.
4. Difficulty in getting a completely built version of the libraries discouraged further use.

There were many good points to both libraries. These were incorporated into what was to become the Naval Postgraduate School Class Library (NPSCL). NPSCL is a collection of stand-alone classes. The classes are either concrete data types such as string, date and time, generic abstract data types for containers (lists, tables, trees) or cooperative classes comprising a support system. At the time that NPSCL was implemented, most C++ compilers did not support templates as defined in the proposed standard and [12]. In order to implement generic templated containers, NPSCL uses macro substitution. This is not the most desirable solution but does not require special preprocessor support and is fairly easy to use. As C++ compilers supporting the C++ template facility become available, we will port NPSCL to true templates.

Run-Time Type Information

NPSCL provides several support systems that NPSGDL uses extensively. The first is a simple run-time typing system. Run-time type information (rtti) was not originally a part of C++ due to the added overhead such a system would impose. A recent proposal [13] to add rtti has been put forth by the language's designer which should correct a major deficiency. Many C++ applications do not need run-time typing support relying instead on virtual methods and dynamic binding of method calls. This falls apart in systems like NPSGDL. All tokens behave similarly and respond to the same messages and can thus be managed generically as tokens. There are many occasions when some specific behavior not common to all tokens is needed from a token. Also there are occasions when certain tokens must be separated from the rest. Both of these situations demand a consistent way to identify the type of object referenced and safely cast pointers down the inheritance hierarchy. This is the purpose of the run-time type system of NPSCL. The type system is non-intrusive meaning that not all classes must participate, although there is little reason not to include all classes. The system is based on that described in [14] and similar to the system described in [13]. Basically each class has a public static data member that contains a string identifier for the class and a list of immediate base classes for the class. Methods and macro support allow the user to query the type of a class, determine if a down-cast is safe, compare for type equality and other helpful functions. This system is useful and effective. It imposes little space overhead on client classes and very little performance overhead by using inline methods where possible. Also since run-time type inquiries are the exception rather than the norm, the system is not a factor in most performance studies conducted. In addition to the string name based type system described, NPSCL provides a simple object identification capability based on integers. Using this system each class instance is given a unique integer identity that can be used for more refined identity testing. These integer identifiers are used by NPSGDL for various Silicon

Graphics, Inc. GL functions [8] among other things.

Reference Counting

Another system extensively used by NPSGDL is a simple reference counting garbage collection system. In order to save space and improve efficiency, many token objects are shared between each other and NPSobjects. For example, consider two NPSobjects. Each is associated with a description file that defines a “gold” Defmaterial. We only want one copy of the Defmaterial but each NPSobject must contain a copy in the event that the NPSobject must write itself to a file. In this case, the first Defmaterial read would allocate space and insert a pointer in a table. The second NPSobject would get a pointer to the first instance rather than a new one. To support this with a minimum of developer worry, NPSCL provides a simple reference counting system using smart pointer objects. This system is based on the examples in [15] and are similar to the “letter-envelope idiom” in [16]. Basically every class in the NPSGDL system contains a reference count member and methods to increment and decrement the count. If an object’s reference count decrements to zero then its memory is returned to the memory system. The management of the reference counts is the responsibility of a friend class that encapsulates a pointer to the referenced class. This class is generic and uses templates to provide type safety. The pointer class, called a Refptr in NPSCL, overloads operators to behave as a normal pointer with the addition of adjusting reference counts as pointers are assigned, copied and destroyed.

There are several advantages and disadvantages to using this system. Many relating to garbage collection in general:

Advantages

1. System is simple and easy to use.
2. Frees developer from many memory management chores.
3. System is non-intrusive. It can be used or not used as desired.

Disadvantages:

1. System imposes overhead on pointer manipulation.
Little overhead is imposed for pointer use or access.

2. User can break system by mixing real pointers and Refptrs.
3. System doesn’t detect circular references that might result in unrecoverable memory.

Despite these disadvantages, this system is used in NPSGDL to great success. In normal use, the circular reference problem is not encountered and the overhead is only noticed at non-critical times such as object creation/initialization. Other phases of an object’s use normally involve accessing the object pointed at and this operation has little to no overhead due to the use of inline methods. The primary limitation to using this garbage collection system is placed on the developer. The developer must ensure that real pointers are not mixed with the smart pointers across scopes. This is to prevent the system from deallocating an object still referenced by a real pointer. This is not a problem for the typical user as all object management is taken care of within NPSGDL. It is a consideration for developers of tools and those managing custom collections of NPSGDL tokens.

The Persistence Model

A primary requirement for NPSGDL is the ability to store and retrieve object definitions to secondary storage. Since C++ does not provide a standard persistence mechanism, one was designed into NPSGDL. The persistence system was modeled after several different systems, in particular the “virtual constructor” methods outlined in [16]. The primary responsibility for storage and retrieval is distributed among the NPSGDL token classes. The most derived class controls most of the process. Each token implements three methods, the read_from, store_on and creator methods as well as a special “reader” constructor called with an input stream. Each takes as input either an input or output stream. The read_from method expects the stream pointer to be located immediately after the typename of the token and reads all fields on the stream applicable to itself until either an ending flag is encountered, as in Deftokens, or a certain number of lines have been read, as with Defpolys. The store_on method stores the token typename and data values in the correct format on

a specified stream. This method has a parameter that tells the token whether it should output its typename with its data. Using this parameter, derived classes can have super classes output their data without inserting extraneous typenames on the stream. The creator method is a static method. This is important because it does not work on a per object basis. The creator method's function is to allocate a new instance of a token and initialize it from an input stream. The creator then returns a pointer to the new token to the caller. The creator method acts as the virtual constructor as explained below.

Using these methods, each token provides facilities to read, write and initialize itself from file streams. Still there must be some object or process in overall control of all this. All of the typenames currently valid in the system must be known in order to recognize them on an input stream. Also, once a typename is recognized, there must be a way of telling the correct token to initialize itself from the stream. This responsibility is delegated to the Token base class. The Token class contains a static table of Tokeninfo objects called the tokentable. A Tokeninfo object associates a token's typename and the address of its creator method. The Token class provides a public static method called `read_token()` that will process an input stream using the following general algorithm:

1. Read a string, assume it's a token identifier
2. Lookup the identifier in the tokentable.
3. If the identifier is valid then get the tokeninfo object for that token, else issue an error and return.
4. Call the identified tokens creator method passing the input stream in.
5. The creator will return a pointer to a valid token initialized from the stream. Return the token pointer to the caller.

The `read_token` method is called repeatedly by the `NPSObject::read_from` method until the end of file is reached.

An important consideration in the design of NPSGDL was the initialization of the tokentable. One of the primary design goals was to be able to add new tokens to the language system with minimum effect on other modules. The ideal being

providing a header (.h) and implementation (.C) file for the new token(s) and adding the object module to the library archive. This ideal is very close to being met. The tokentable is initialized dynamically using a special constructor in each token and a special instantiation of each token.

The special constructor is one that takes as its sole argument a Tokeninfo object. Remember the Tokeninfo object is the object placed into the tokentable. The base class Token's special constructor places the Tokeninfo object into the table. All derived tokens simply call their base class's special constructor passing the Tokeninfo object along to the root of the hierarchy.

In order for this system to work, this constructor must be used. Thus in each token's implementation file, a single static object of the particular token type is instantiated using the special constructor. The C++ language guarantees that global static objects will be constructed before `main()` is entered. So all of the static objects used for tokenable initialization are constructed prior to `main()` ensuring that the tokentable is properly initialized automatically at run-time. There is no need for user initialization.

This system for dynamic initialization works very well. There is one problem though, current linker technology does not support this model well. Typical NPSGDL users use only the `NPSObject` class and don't directly refer to the token subclasses. Since the `NPSObject` classes deal with tokens in the abstract through Token pointers, it does not refer to derived tokens directly either. Thus there are normally no references to derived token modules for the linker to resolve. The result is that the linker does not include the modules for derived classes and the tokentable is not initialized properly.

The temporary remedy was to fall back to the more traditional method of having an object or module that ensures that each module is linked in. Instead of having an external class manage the tokentable initialization, we have an external object, called a `TokenRegistrar`, whose constructor calls a static method named `register_token()` provided by each token class. This method does nothing. It is used solely to generate an unresolved reference

for the linker. The TokenRegistrar is instantiated in the Token module which is always linked in. The advantage to this approach is that the registration functionality is easily removed without affecting anything.

The long term remedy is more intelligent linker technology. Dynamic linking to the degree needed is not supported on the platforms we use. Shared libraries don't address the problem either and are very difficult to construct for a system such as NPSGDL. As object oriented programming and C++ become more popular, linker technology must improve to support the highly dynamic designs possible like this initialization scheme.

There is another consequence to using the registration method. NPSobjects only know about the class Token. The actual derived tokens used are instantiated at run-time and accessed via dynamically bound calls. Thus there is no way to know which tokens will be needed by any set of definition files. Therefore all token modules must be linked to the application. This results in large applications with possibly a lot of unused code. The alternative is to preprocess definition files and only link the modules needed. However, this is very limiting and establishes an application dependence. The definition files can't be modified without preprocessing again. While this option may be offered in the future, it is not generally acceptable.

Using NPSGDL

The normal usage of NPSGDL is very straight forward and simple. The NPSobject class is the normal interface to the system. The user simply instantiates an NPSobject with a definition file name, defines the NPSobject to ensure that any Deftokens are defined to the graphics system and displays the object when desired. This technique is illustrated in Figure 2. In addition to this object definition-file based approach, the user may instantiate and use individual tokens. This way the NPSGDL tokens can be used as individual abstractions of the underlying graphics library. A simple example of this is shown in Figure 3. The key point is that normal use of NPSGDL is simple

```
// program display_planets.C
#include "NPSobject.H"
#include "gl.h"
void init()
{
    // open and config a gl window
    winopen("planets");
    RGBmode();
    doublebuffer();
    gconfig();
}

main() {
float bgcolor[4] = {0.0, 0.0, 0.0, 0.0};
NPSobject p_obj("planets.off"); // read the file

init();
p_obj.define(); // define any Deftokens

while(1) { // kill from win manager
    c4f(bgcolor);
    clear(); // clear the window
    // put up viewing and transformation stuff
    p_obj.display();
    swapbuffers();
}
}
```

Figure 2

since management of low level details is hidden from the user. As mentioned before, NPSGDL provides a standard method of describing graphical objects. Since a large number of objects exist in our laboratory, few developers need to worry about the details of creating description files. Many simply use what's available thereby speeding the prototyping and development process.

Performance Considerations

NPSGDL is used in complex, interactive, real-time 3D graphics applications. Performance is a critical issue. Limited analysis has shown that NPSGDL is very efficient when compared to previous systems with similar purposes [2]. NPSGDL does impose a small amount of overhead over managing objects manually using traditional

```

// program use_materials.C
#include "NPSObject.H"
#include "Setmaterial.H"

void init()
{
    // open and config a gl window
    winopen("planets");
    RGBmode();
    doublebuffer();
    gconfig();
}
main() {
float backcolor[4] = {0.0, 0.0, 0.0, 0.0};

// read the file containing material definitions
NPSObject m_obj("allmaterials.off");
init();
m_obj.define(); // define all Deftokens (materials)

while(1) { // kill from win manager
    c4f(backcolor);
    clear(); // clear the window
    // put up viewing and transformation stuff
    Setmaterial cur_mat("gold");
    // draw something that is gold
    swapbuffers();
}
}

```

Figure 3

methods. This is primarily due to the tradeoffs between designing for application use and general tool use. Many of the data structure classes used support high level operations that will be helpful to NPSGDL tool designers. While this does not imply that the data structures are inefficient, it does increase the code size. A good example is the use of a string class that manages memory and supports substrings and concatenation over typical C character pointers. But as the objects become complex, the overhead of NPSGDL factors out. Also the flexibility and utility of the system compensates for any minor performance degradations.

NPSGDL performance is based on two main strategies. First all calculations are preprocessed

prior to display. Second routines that are called often or are demonstrated bottlenecks are optimized based on profiler feedback.

The first strategy has the most impact. Where possible, each token performs any calculations needed either during initialization or definition. Also graphics data is cached for fast retrieval rather than calculated during display. This is the classic trade of space for speed. For example, during initialization, the Defsphere token calculates the vertices for a tessellated sphere and stores the results in an array for easy access. This technique is not new but important and applicable.

Another performance boost is gained in the NPSObject class. Each NPSObject contains five lists of Tokens. The main list contains all tokens and is the basis for the other lists. The main list is used for reading and writing. The second list contain only Deftokens, while the other lists are used for display and contain tokens of the same resolution. This approach reduces the number of function calls required at different phases of object use. Remember that each token responds to a common set of messages. If the token does not directly handle a message, for instance a Deftoken does not implement a display() method, then the default method is invoked in a base class, typically Token. This default method is usually empty but still imposes a function call. These unnecessary calls are prevented by using specific lists at the expense of space and preprocessing time.

The second strategy involves the normal performance tuning cycle. Profilers were used to identify those functions that were called the most and those that used the highest percentage of cpu time. These were then analyzed for optimization.

The primary method during the first optimization phase was the use of inline functions. This came into play in two ways. The first way is the normal use of the C++ inline construct. During design, only the most simple methods were designated as inline, reserving it's use for tuning. This was to prevent code size growing to the point of generating excess page faults in the virtual memory system. This proved very prudent as additional inlining was not needed very much. Functions that were called fairly often that were candidates for

inlining were made inline. The page fault rate was monitored for excess increases as a result of the inlining. The second inlining method used was to reduce or eliminate member function calls from within a class. While the function calls make the code easier to read and more compact, they result in one or more function calls in order to accomplish something with data already accessible to the first member function. So, where feasible, each member function manipulates class data explicitly rather than through other member functions. This is also true in the case where the other member function is inline. Since the inline directive can be ignored by the compiler, the inline expansion was used directly when needed.

After analyzing critical functions, some general and machine specific optimizations were made [17]. One of the first was to unroll loops during display. For example, a Defpoly contains a list of n vertices. If n is small, less than seven, the traversal loop is unrolled yielding better performance. Also pointer arithmetic is used over subscripting when it is safe and convenient to do so. One of the machine specific techniques used is to use four element float arrays for vertices rather than three elements. This is due to the read length of the cache system in Silicon Graphics Powervision™ series workstations.

NPSGDL drawing performance was compared to that of NPSOFF [2], a system with similar behavior implemented in C. The comparison consisted of displaying the same object 10000 times and reporting the average user and system time to display a single frame. The object displayed was a gouraud shaded, lit cube with two textured sides, a small line and a simple decal of a triangle on a square. The texturing was done with both explicit coordinates and automatically generated texture coordinates. The object was rendered on a Silicon Graphics, Inc. Iris 4D/340 VGX using a perspective projection and z-buffered hidden surface removal. The results of five runs of each timing program are shown below with user time over system time in microseconds:

SYSTEM	1	2	3	4	5	Avg.
NPSOFF	1438	1491	1501	1645	1629	1541
	900	828	923	870	919	888
NPSGDL	1466	1371	1432	1470	1572	1462
	874	932	924	1033	894	931
Avg. Total:						
NPSOFF	2429					
NPSGDL	2393 1.4% difference					

These results are encouraging. While the two systems share common capabilities, NPSGDL provides many additions and improvements over NPSOFF. In addition to the benefits gained from the object-oriented design and implementation, NPSGDL provides extensive error detection and recovery, reasonable default behavior, extended data structure support, name space control and more high level application support.

Limitations, Future Work and Conclusions

NPSGDL is the most recent effort to provide application developers with an easy to use, application independent method to describe, store, share and manipulate graphical objects. NPSGDL does this well. However there are limitations to the system. The components and focus of NPSGDL is still relatively low-level and platform specific. Many of the NPSGDL tokens correspond closely to Silicon Graphics GL functions [8]. This still requires the developer to understand the use and interactions of many GL functions. A higher level of abstraction, including more high level components, would be helpful in many cases.

Another limitation is the size of the NPSGDL library. As mentioned before, the typical linker must be forced to link the modules of the derived token classes. Since it is not known prior to execution which tokens will be needed, all derived tokens and their support modules must be linked to an application. This results in almost the entire library being linked to an application even though many tokens may not be used. This is more of a linker problem than an NPSGDL problem but is still a consideration for developers, as the size of the resulting executable and the placement of token modules may affect the page fault and cache

performance of the application.

Future work on NPSGDL will try to address these issues. The primary focus of future work will be to add extended functionality. Sound support is important to development at NPS. Once a standard, consistent mechanism for sound generation is in place, NPSGDL will likely support it. Another important aspect of many objects within NPS applications is physical characteristics like mass, center of gravity, etc. Extensions to NPSGDL that embody object characteristics for supporting physically based modeling are also being considered. Other high level graphics support for things like atmospheric effects, motion blur, and anti-aliasing are also likely candidates for support by NPSGDL.

NPSGDL provides high level application support. It incorporates an application independent language for describing graphical objects as well as a medium to high level graphics interface system. It is simple to use, easy to extend and maintain, and very flexible and capable. NPSGDL gives application designers the leverage needed to rapidly prototype and develop applications. The ability to create and maintain collections of object models as well as individual components that can be used in many different systems is critical to virtual world development at NPS.

Acknowledgments

We wish to acknowledge the sponsors of our efforts, in particular Major David Neyland, USAF of DARPA/ASTO, George Lukes of the USA Topographics Engineer Center, Michael Tedeschi of the USA Test and Experimentation Command, John Maynard and Duane Gomez of the Naval Ocean Systems Center, San Diego, LTC Dennis Rochette, USA of the Headquarters Department of Army AI Center, Washington, D.C., Carl Driskell of PM-TRADE and the Naval Postgraduate School Direct Funding Program.

References

1. Zyda, Michael J., Pratt, David R., Monahan, James G. and Wilson, Kalin P., "NPSNET: Constructing a 3D Virtual World", Proceedings of the 1992 Symposium on 3D Interactive Graphics, March 1992.

2. Wilson, Kalin P., Zyda Michael J., Pratt, David R., and Monahan, James G. "NPSOFF: An Object Description Language for Supporting Virtual World Construction", in submission.

3. Wisskirchen, Peter, Object-Oriented Graphics, Springer-Verlag, 1990.

4. Wisskirchen, Peter, Object and Constraint Paradigms for Graphics, "Object-Oriented and Classical Approaches", ACM SIGGRAPH 1991 course 22 notes, 1991.

5. Egbert, Parris K. and Kubitz, William J. "Application Graphics Support through Object-Orientation, November 14, 1991, draft paper.

6. Zelenik, Raobert C., Conner, D. Brookshire, Wloke, Mathias M., Aliaga, Daniel G., Huang, Nathan T., Hubbard, Philip M., Knep, Brian, Kaufman, Henry, Hughes, John F., van Dam, Andries, "An Object-Oriented Framework for the Integration of Interactive Animation Techniques", ACM Computer Graphics, vol 25 number 4, Siggraph '91 Proceedings, pp. 105-111, 1991.

7. Wirfs-Brock, Rebecca, "Responsibility Driven Design"

8. Silicon Graphics Computer Systems, Inc., Graphics Library Reference Manual, C edition, Iris-4D series, 1990

9. Magnenat-Thalman, Nadia, Thalman, Daniel, Computer Animation, Theory and Practice, second edition, pp161-162, Springer-Verlag, 1990.

10. Gorlen, Keith E., Orlow, Sanford M., Plexico, Perry S., Data Abstraction and Object-Oriented Programming in C++, John Wiley & Sons, 1990.

11. The Texas Instruments C++ Object-Oriented Library Users Manual, Texas Instruments Inc., 1990.

12. Ellis, Margaret A. and Stroustrup, Bjarne, The Annotated C++ Reference Manual, Addison-Wesley, 1990.

13. Stroustrup, Bjarne and Lenkov, Dmitry, "Runtime Type Identification for C++, A Proposal for New Features to the Language.", The C++ Report, vol. 4 num. 3, pp. 32-42, Mar-Apr 1992.

14. Stroustrup, Bjarne, The C++ Programming Language, Second Edition, Addison-Wesley, 1991.

15. Shapiro, Jonathan. A C++ Toolkit, Prentice Hall, 1991.

16. Coplien, James O., Advanced C++, Programming Styles and Idioms, Addison-Wesley, 1992.

17. Silicon Graphics Computer Systems inc., Graphics Library Programming Tools and Techniques, pp 2-1 - 2-53, 1991.