3 Geometry

3.1 Introduction

Having established an architecture in the previous chapter, which creates, manages and renders geometric primitives, we can now consider the actual implementation of those primitives. All primitives derive from a virtual base class, which has been defined as:

```
class Primitive {
public:
  virtual bool boundable();
  virtual Bound bound();
  virtual bool splitable();
  virtual void split(Framework &f, bool usplit, bool vsplit);
  virtual bool diceable(MicroPolygonGrid &g, Hider &h,
      bool &usplit, bool &vsplit);
  virtual void dice(MicroPolygonGrid &g);

protected:
  virtual Point evalP(float ugrid, float vgrid);
  virtual Vector evaldPdu(float ugrid, float vgrid);
  virtual Vector evaldPdv(float ugrid, float vgrid);
};
```

Creating specific forms of geometric primitive is therefore simply a case of filling in the implementation of these member functions. For each class of primitive we must consider bounding, splitting and dicing.

3.2 Parametrics

The most common and simplest class of surfaces are the parametrics. These are defined by an equation:

$$P = f(u,v)$$

where u and v are usually restricted to be in the range 0–1. If we make the additional assumption that the function defining the surface is generally well

behaved (continuous, and bounded), as the standard parametric surfaces used for rendering are, then we can create a generic parametric class which can implement most of the required functions, minimizing the work which must be done for each individual parametric surface.

3.2.1 A Generic Parametric Surface

Bounding

All of the standard parametric surfaces are boundable, and therefore the base class can define `boundable()` to return true. However, while it is possible to approximate the bounds in a generic fashion, `bound()` is best implemented on a per primitive basis. In fact for most parametric surfaces bounding is the most difficult function to implement.

Splitting

While certain forms of parametric surface may benefit from a more specific implementation, the splitting of parametric surfaces can be handled simply and efficiently in a generic fashion. Four member variables can be used to define the area of the original patch that the new sub-patch represents:

```
float umin, umax
float vmin, vmax
```

Rather than define the surface over the ranges $u = (0, 1]$, $v = (0, 1]$ the new ranges $u =$ (umin, umax], $v =$ (vmin, vmax] are used instead. By default patches are created with these variables initialized to 0 and 1, so they represent the whole patch.

Patches can now trivially be split in u:

```
if(usplit)
  {
  Paramtric *left=duplicateSelf();
  Paramtric *right=duplicateSelf();
  float umid= (umin+umax)*0.5f;
  left->umax=umid;
  right->umin=umid;
  f->insert(left);
  f->insert(right);
  }
```

and similarly in v.

In addition to being generic, this approach is far simpler and faster, than most geometry specific patch splitting routines. If the `duplicateSelf()` member function is implemented using reference counting to a shared set of parameters,