Using Projective Space to Improve Precision of Geometric Computations - Appendix

Krzysztof Kluczek, Gdańsk University of Technology
krzych82@poczta.onet.pl

Introducing Third Dimension

This document presents an extension of algorithms and ideas found in the book, which allows them to be applied to geometry in 3D space.

Just like $\mathbf{RP}^2$ projective space can be used to perform operations on objects in $\mathbf{R}^2$ space, $\mathbf{RP}^3$ projective space can be used to perform geometric computations in $\mathbf{R}^3$ space. Like $\mathbf{RP}^2$ space can be understood as a space of lines in $\mathbf{R}^3$ passing through the origin, $\mathbf{RP}^3$ projective space can be seen as a space of lines passing through point $[0,0,0,0]$ in $\mathbf{R}^4$ space. As four-dimensional space can't be easily imagined, we can exploit similarities of $\mathbf{RP}^2$ and $\mathbf{RP}^3$ projective spaces and extend operations in $\mathbf{RP}^2$ space by adding an extra dimension. Because of this extra dimension, all vectors used for referring elements of $\mathbf{RP}^3$ space will be four-component vectors $[x, y, z, w]$. Just like it was the case in $\mathbf{RP}^2$ projective space, a pair of vectors $\mathbf{P}$ and $\mathbf{Q}$ in $\mathbf{RP}^3$ space will refer to the same element (a line passing through the origin) of this space if and only if $\mathbf{P} = \mathbf{Q}c$ for certain scalar $c \neq 0$. As all lines pass through the origin, the vector $[0,0,0,0]$ is of no use while referring to elements of $\mathbf{RP}^3$ space. When $\mathbf{RP}^2$ projective space was used for operations on primitives in $\mathbf{R}^2$ space these spaces were related to each other by perspective transformation onto the $z = 1$ plane with the center of this perspective transformation at the origin. Almost the same is true for operations in $\mathbf{RP}^3$ space presented here. To relate a vector in $\mathbf{RP}^3$ projective space to a point in $\mathbf{R}^3$ space we can use perspective projection onto $w = 1$ hyperplane with the center of projection placed at the origin. A vector $[x, y, z, w]$ with $w \neq 0$ in $\mathbf{RP}^3$ projective space will represent a point $[x/w, y/w, z/w]$ in $\mathbf{R}^3$ space. This representation of a point in $\mathbf{R}^3$ with a vector in $\mathbf{RP}^3$ may be familiar to some readers as it is widely used in computer graphics. However, the motivation for using such space is different, as it is usually used mainly for performing transformations on point data using matrices, while here it will be used to improve precision of geometric computations.

Basic Objects in $\mathbf{R}^3$

Just like in two-dimensional space $\mathbf{R}^2$ we can focus on points and directed lines, in $\mathbf{R}^3$ space we will perform operations on points and directed planes. Directed plane can be described as a plane with its one side defined to be a positive side and the other side to be the negative one. This extension is similar to extension of lines into directed lines and allows for efficient definition of solids. For example, in case of defining boundary of a solid directed planes used can be placed in such way, that their positive sides face the interior of the solid. Other objects in $\mathbf{R}^3$ space can be described using points and directed planes. For example, a line in $\mathbf{R}^3$ space can be defined as an intersection of two planes.

Points and Directed Planes in RP^3

As it was stated earlier, a point $[s,t,r]$ in $\mathbf{R}^3$ space will be represented by a four-dimensional vector in $\mathbf{RP}^3$ projective space. The $w=1$ hyperplane is used for perspective transformation from $\mathbf{RP}^3$ projective space to $\mathbf{R}^3$ space, so the simplest way to find the representation of a point from $\mathbf{R}^3$space in $\mathbf{RP}^3$ space is to assume $w=1$, which results in a vector $[s,t,r,1]$ representing this point. To transform a vector $[x,y,z,w]$ in $\mathbf{RP}^3$ space representing a point back to $\mathbf{R}^3$ space we can use perspective transformation onto $w=1$ hyperplane, which results in a point $[x/w, y/w, z/w]$ in $\mathbf{R}^3$ space. Again, like it was the case with operations in $\mathbf{RP}^2$ space, we will assume $w>0$ for simplicity of further computations. If it's not the case, we can scale entire vector by -1 to fix this. Vectors with $w=0$ don't represent valid points in $\mathbf{R}^3$ because of perspective transformation used. Functions used to find representations of points in $\mathbf{R}^3$ and $\mathbf{RP}^3$ spaces are shown below as Equations 1 and 2. Note that these functions operate on elements of $\mathbf{R}^4$ space as they take an argument or result in a vector from $\mathbf{R}^4$ space referring to one of the elements (lines passing through the origin) of $\mathbf{RP}^3$ projective space.

$$f:\mathbf{R}^3 \to \mathbf{R}^4, \quad f([s,t,r]) = [s,t,r,1] \tag{1}$$
$$g:\mathbf{R}^4_{w\neq 0} \to \mathbf{R}^3, \quad g([x,y,z,w]) = [x/w, y/w, z/w] \tag{2}$$

The other object of particular interest in $\mathbf{R}^3$ space is a directed plane. In $\mathbf{RP}^2$ projective space a directed line was represented by a directed plane crossing the origin, which intersection with $z=1$ plane resulted in a line being represented. By adding an extra dimension, we can represent a directed plane in $\mathbf{R}^3$ with a directed hyperplane crossing the origin, which intersection with $w=1$ hyperplane will result in a plane being represented. As the operations in four-dimensional space can't be easily imagined, to fully understand this definition we can take a look at math behind it. The hyperplane in $\mathbf{R}^4$ space can be defined using its normal and its distance from the origin just like a plane in $\mathbf{R}^3$ with this exception, that the normal is a four-dimensional vector. The equation of a hyperplane in $\mathbf{R}^4$ space is shown below as Equation 3.

$$[x,y,z,w]\cdot\mathbf{N} + d = 0 \tag{3}$$

which can be written as:

$$xN_x + yN_y + zN_z + wN_w + d = 0 \tag{4}$$

Because we consider only hyperplanes crossing the origin, we the displacement distance $d=0$ for all hyperplanes used except $w=1$ the hyperplane used for perspective projection. Therefore, the hyperplane equation is reduced to a four-dimensional dot product and such hyperplane can be represented using only its normal. We can drop the d component from the Equation 4 in this case. Also, because only vectors with $w > 0$ will be used for representations of points in $\mathbf{RP}^3$ space, dividing both sides of the equation by w gives following result:

$$\frac{x}{w}N_x + \frac{y}{w}N_y + \frac{z}{w}N_z + N_w = 0 \tag{5}$$

Any vector $[x, y, z, w]$ in $\mathbf{RP}^3$ space represents a point $[x/w, y/w, z/w]$ in $\mathbf{R}^3$ space, so the Equation 5 is an equation of a plane in $\mathbf{R}^3$ space defined for vectors representing points in $\mathbf{RP}^3$ space. To define positive and negative sides of a directed plane we can use the sign of the left sides of equations presented above. The point represented by vector $[x, y, z, w]$ with $w > 0$ can be defined to lie on the positive side of the plane if and only if $[x, y, z, w] \cdot \mathbf{N} > 0$ for this point. The negative side of the plane can be defined similarly.

Cross Product of Four-dimensional Vectors

While the definition of dot product can be easily extended from three to four dimensions, it's not the case with the cross product operation. Fortunately, extending the definition of cross product in $\mathbf{R}^3$ space given in the article in the book with additional dimension results in operation sharing many characteristics with the cross product in $\mathbf{R}^3$ space ([Hollash91]). Cross product in $\mathbf{R}^4$ space can be defined as an operation taking three arguments which are vectors from $\mathbf{R}^4$ space. Additional row of basic unit vectors is added to form a 4x4 matrix and the result of the cross product in $\mathbf{R}^4$ space can be computed as a determinant of this matrix. This operation is shown as Equation 6 below.

$$\mathbf{P} \times \mathbf{Q} \times \mathbf{R} = \begin{bmatrix} P_x & P_y & P_z & P_w \\ Q_x & Q_y & Q_z & Q_w \\ R_x & R_y & R_z & R_w \\ \mathbf{i} & \mathbf{j} & \mathbf{k} & \mathbf{l} \end{bmatrix} \tag{6}$$

This definition can be expressed in a more readable form by expanding the determinant using its last row to the form shown below as Equation 7.

$$\mathbf{P} \times \mathbf{Q} \times \mathbf{R} = \begin{vmatrix} P_y & P_z & P_w \\ Q_y & Q_z & Q_w \\ R_y & R_z & R_w \end{vmatrix} \mathbf{i} - \begin{vmatrix} P_x & P_z & P_w \\ Q_x & Q_z & Q_w \\ R_x & R_z & R_w \end{vmatrix} \mathbf{j} + \begin{vmatrix} P_x & P_y & P_w \\ Q_x & Q_y & Q_w \\ R_x & R_y & R_w \end{vmatrix} \mathbf{k} - \begin{vmatrix} P_x & P_y & P_z \\ Q_x & Q_y & Q_z \\ R_x & R_y & R_z \end{vmatrix} \mathbf{l} \tag{7}$$

Four-dimensional cross product shares many characteristics with standard cross product. Most importantly, the resulting vector is perpendicular to all three input vectors, which can be easily proven by computing a dot product of the result and any of the input vectors of the cross product in $\mathbf{R}^4$ space, which is always zero. The other fact about this operation is that if any two input vectors are linearly dependent, the resulting vector is $[0,0,0,0]$, as the linear dependence of input vectors introduces linear dependence of certain rows in determinants being computed, which causes all resulting vector coordinates to be zeroes. The last important fact about this operation is that swapping any two input vectors the cross product in $\mathbf{R}^4$ space results in a change of sign of all resulting vector coordinates, as swapping any two rows of a determinant changes its sign. To sum it up, the cross product in $\mathbf{R}^4$ space has basic characteristics of the cross product in $\mathbf{R}^3$ space, with main difference that in operates on vectors from $\mathbf{R}^4$ space and unlike standard cross product requires three arguments instead of two.

As you can see from Equations 6 and 7, computing four-dimensional cross product is quite complex task requiring many operations to compute each determinant. Therefore, a small optimization to Equation 7 can be done to compute the result more effectively. Expanding each determinant from Equation 7 by its last row results in following formula:

$$\begin{aligned}\mathbf{P}\times\mathbf{Q}\times\mathbf{R} = &\left(\begin{vmatrix} P_z & P_w \\ Q_z & Q_w \end{vmatrix} R_y - \begin{vmatrix} P_y & P_w \\ Q_y & Q_w \end{vmatrix} R_z + \begin{vmatrix} P_y & P_z \\ Q_y & Q_z \end{vmatrix} R_w \right)\mathbf{i} \\ &- \left(\begin{vmatrix} P_z & P_w \\ Q_z & Q_w \end{vmatrix} R_x - \begin{vmatrix} P_x & P_w \\ Q_x & Q_w \end{vmatrix} R_z + \begin{vmatrix} P_x & P_z \\ Q_x & Q_z \end{vmatrix} R_w \right)\mathbf{j} \\ &+ \left(\begin{vmatrix} P_y & P_w \\ Q_y & Q_w \end{vmatrix} R_x - \begin{vmatrix} P_x & P_w \\ Q_x & Q_w \end{vmatrix} R_y + \begin{vmatrix} P_x & P_y \\ Q_x & Q_y \end{vmatrix} R_w \right)\mathbf{k} \\ &- \left(\begin{vmatrix} P_y & P_z \\ Q_y & Q_z \end{vmatrix} R_x - \begin{vmatrix} P_x & P_z \\ Q_x & Q_z \end{vmatrix} R_y + \begin{vmatrix} P_x & P_y \\ Q_x & Q_y \end{vmatrix} R_z \right)\mathbf{l}\end{aligned} \tag{8}$$

Each 2x2 determinant in Equation 8 is used exactly twice, so it's sufficient to compute each of these determinants just once and reuse the result, saving fair amount of computations.

Basic Operations in RP^3

Basing on our experience with representation of $\mathbf{R}^2$ space with $\mathbf{RP}^2$ projective space, we can define three basic operations in $\mathbf{R}^3$ space, which will be performed in $\mathbf{RP}^3$ space. Being given an ordered triple of points we can find a directed plane passing through them in such way, that from positive side of the plane they can be seen in clockwise order. Being given a point and a directed plane we can determine which side of the plane the point lies on or whether it lies on the plane. Finally, being given three directed planes we can find their intersection point.

The first basic operation we consider is finding a directed plane passing through three given points. We are given three vectors representing these points in $\mathbf{RP}^3$ space. In this space a plane in $\mathbf{R}^3$ space is represented by a hyperplane crossing the origin of the projective space and the normal of this hyperplane should be computed. Because the hyperplane we are looking for passes through the origin, it will contain the projections of given vectors onto $w = 1$ plane if and only if it will contain these given vectors. Therefore, the hyperplane normal has to be perpendicular to all three given vectors, which makes four-dimensional cross product a perfect tool for finding this normal. The four-dimensional cross product has the property, that changing order of any two input vectors changes sign of all coordinates of resulting normal, which allows controlling which side of the hyperplane the normal will be facing. We can define positive side (the side the normal is pointing at) as a side, from which points making the plane are visible in clockwise order, but as this depends on the handedness of used coordinate system, it should be checked during implementation of this operation and two of input vectors in cross product should be swapped if needed. It’s also possible that input points are collinear, so a single plane passing through them can’t be computed. Such case can be easily detected as it causes the resulting normal to be $[0,0,0,0]$.

The next operation of particular interest is checking which side of the directed plane given point lies on or whether it lies on this plane. As usual, the plane is represented in $\mathbf{RP}^3$ projective space with a hyperplane. According to the definition of the hyperplane in $\mathbf{RP}^3$ projective space, the sign of a dot product of hyperplane normal and a vector representing the point in $\mathbf{RP}^3$ space can be used for this check. We have to make sure that all points being tested have component $w > 0$, because as all elements (lines) of $\mathbf{RP}^3$ projective space cross at the origin, the sign of the dot product and the result of the test will be inversed for vectors with negative w components (note that it’s still perfectly legal for normal of the hyperplane to have $w > 0$). Fortunately, we have already placed the $w > 0$ restriction on vectors representing points. Therefore, positive result of the dot product indicates that the point lies on the positive side of the plane. Likewise, negative result of the dot product indicates that the point lies on the negative side of the plane and if the point lies on the plane, the result of the dot product is zero.

The last operation we define is an operation, which allows us to find intersection point of three given planes. As vector representing intersection point in $\mathbf{RP}^3$ space has to lie on all three hyperplanes representing the planes being given, its dot products with normal of each hyperplane should be zero. Again, four-dimensional cross product comes in handy as this operation on hyperplane normals will result in a vector satisfying above requirement. Because resulting vector lies on all hyperplanes, the point in $\mathbf{R}^3$ space represented by this vector will lie on all three planes represented by these hyperplanes, being their intersection point. To satisfy $w > 0$ requirement for resulting vector coordinates we have to check whether its already true and if it isn’t, all coordinates of the vector have to be scaled by -1 to obtain proper vector representing the point. If resulting vector has coordinate $w = 0$, it means that intersection point can’t be computed because two or more input planes are parallel.

Line Representation in $\mathbf{RP}^3$

The definitions and operations defined above take points and directed planes into account. To be able to perform geometric computations in $\mathbf{R}^3$ space we still need some representation for lines, which will allow performing geometrical operations on them. Making use of already defined objects, a line in $\mathbf{R}^3$ space can be defined as an intersection of two planes. We can perform a number of operations on lines defined this way. For example, finding an intersection of a line and a plane is straightforward, as line-plane intersection point is just an intersection point of the plane and two planes used to define the line. Similarly, to check whether a point lies on the line or not we can perform point-plane check for each plane defining the line, as a point lies on the line if and only if it lies on both these planes at the same time.

The most important operation, when it comes to lines, is finding a line passing through a pair of given points. As we have only two points, two extra points have to be picked in such way, that used with two points already present they can define two planes, which intersection is the desired line. These points can be picked arbitrarily, but with some checks should be made assuring that resulting planes correctly define the line. The origin $[0,0,0]$, represented by vector $[0,0,0,1]$ in $\mathbf{RP}^3$ space, is the obvious choice for one of these extra points and together with a pair of initial points can be used to create a first plane. After the hyperplane representing the first plane of the line was constructed we should check whether this plane is a correct plane. If initial points are collinear with the origin, the resulting hyperplane will be invalid and will have normal vector $[0,0,0,0]$. In this case different extra point should be picked and the process should be repeated. To create second plane used to define the line we have to pick yet another arbitrary point. This point should be checked against the plane already created. If it lies on this plane it can't be used and we have to pick yet another point to make another attempt in creating the second plane. After the second plane was created we can be sure that it's a correct plane. The second picked point can't be collinear with two points defining the line, because otherwise it would lie on the first created plane, which we have already checked. We can use following sequence of extra points to be checked: $[0,0,0]$, $[1,0,0]$, $[0,1,0]$, $[0,0,1]$. This sequence guarantees that both planes will be eventually found unless they lie in exactly the same place. This sequence is also a simplest sequence that can be used for this operation and this fact can be used to optimize operations used in the algorithm. If the algorithm described here is to be used frequently, special versions of operations used for construction of the planes can be created, as computing four-dimensional cross product of two given vectors and one of the representations of points from the sequence used ($[0,0,0,1]$, $[1,0,0,1]$, $[0,1,0,1]$, $[0,0,1,1]$) can be optimized by ignoring determinants that are going to be multiplied by zero anyway. Also the dot product operation used to check points from this sequence against the first created plane can be optimized similarly. However, in many cases entire algorithm can be simplified even more, as we often have the first plane already available. For example when lines are used to define edges of a polygon, we can use the plane associated with the polygon as the first plane and we have only to find second planes used to define each of these lines.

Number Range Limits in Geometrical Computations in $\mathbf{RP}^3$

The operations in $\mathbf{RP}^3$ projective space described above can be used to perform precise computations based only on integer math the same way integer math was used for computations in $\mathbf{RP}^2$ space. When using these operations we should also consider numerical range used, as four-dimensional cross product introduces even more multiplications when compared to the standard cross product in three dimensions. To prevent numbers from growing infinitely we again introduce three classes of objects, for which we can estimate the maximum range of integers used at every step of performed operations.

The first class of objects are points with their coordinates in $\mathbf{R}^3$ space given explicitly, for example imported from modeling program or computed using classic geometry in $\mathbf{R}^3$ space. Again, the range of these coordinates influences ranges of all values used in further operations, so the range of these coordinates should be limited. Because we are working on integers only, to perform computations on fractional coordinates, which are often common in input data, we have to scale them up according to required precision and round to nearest integers. Note that this rounding reduces only the precision of input data, but not the precision of further geometric operations itself. The maximum coordinate range after scaling point coordinates up is the main factor influencing ranges of values used during further operations, so a compromise between workspace size, precision of representation of fractional input coordinates and resulting integer coordinate range should be found.

The second class of objects are directed planes computed as planes crossing any tree points from input data. Similarly to the case of representation of directed lines in $\mathbf{RP}^2$ space, the normal of hyperplane representing a directed plane computed this way is a result of single four-dimensional cross product.

The last class of objects being used are points computed as intersections of directed planes of the above class. Again, vectors representing such points in $\mathbf{RP}^3$ space are results of four-dimensional cross products performed on hyperplane normals. These three classes of objects described are sufficient for most geometric algorithms and such algorithms should be created in a way that doesn't require more complicated operations. Most importantly, points computed as intersections of planes can't be used to define other planes. The definition of lines presented earlier isn't included as different class of objects, as it's based on directed planes for which we already defined a class.

To estimate the ranges of values resulting in defined operations, we have first to estimate the range of coordinates of vectors resulting from four-dimensional cross product, as most operations use it. The range of w component of input and resulting vectors is estimated separately as its range will often be different from the range of x, y and z components in input vectors, which in turn can cause the range of w component in resulting vector to be different from ranges of other components. This is especially true for operations on points with their coordinates specified directly, which representations in $\mathbf{RP}^3$ space always have coordinate $w = 1$. Table 3. contains the range analysis of values at various steps of computations of four-dimensional cross product, according to Equation 8.

Table 3. Numerical ranges of values used during computation of four-dimensional cross product.

Object	Equation	Range
Input vector ($\mathbf{P}, \mathbf{Q}, \mathbf{R}$)	x, y, z	$[-a, a]$
	w	$[-b, b]$

2x2 determinant without w coordinate	$\begin{vmatrix} P_x & P_y \\ Q_x & Q_y \end{vmatrix} = P_x Q_y - P_y Q_x$	$\left[-2a^2, 2a^2\right]$
2x2 determinant with w coordinate	$\begin{vmatrix} P_x & P_w \\ Q_x & Q_w \end{vmatrix} = P_x Q_w - P_w Q_x$	$\left[-2ab, 2ab\right]$
2x2 determinant without w coordinate multiplied by one of x, y, z coordinates	$\begin{vmatrix} P_x & P_y \\ Q_x & Q_y \end{vmatrix} R_z$	$\left[-2a^3, 2a^3\right]$
2x2 determinant with w coordinate multiplied by one of x, y, z coordinates	$\begin{vmatrix} P_x & P_w \\ Q_x & Q_w \end{vmatrix} R_y$	$\left[-2a^2b, 2a^2b\right]$
2x2 determinant without w coordinate multiplied by w coordinate	$\begin{vmatrix} P_x & P_y \\ Q_x & Q_y \end{vmatrix} R_w$	$\left[-2a^2b, 2a^2b\right]$
Resulting x, y, z coordinates	$\begin{vmatrix} P_z & P_w \\ Q_z & Q_w \end{vmatrix} R_y - \begin{vmatrix} P_y & P_w \\ Q_y & Q_w \end{vmatrix} R_z + \begin{vmatrix} P_y & P_z \\ Q_y & Q_z \end{vmatrix} R_w$	$\left[-6a^2b, 6a^2b\right]$
Resulting w coordinate	$\begin{vmatrix} P_y & P_z \\ Q_y & Q_z \end{vmatrix} R_x - \begin{vmatrix} P_x & P_z \\ Q_x & Q_z \end{vmatrix} R_y + \begin{vmatrix} P_x & P_y \\ Q_x & Q_y \end{vmatrix} R_z$	$\left[-6a^3, 6a^3\right]$

Having estimated ranges used during four-dimensional cross product operation we can estimate ranges of results of operations used. This analysis is given in Table 4. As it can be seen, even using only three specified classes of objects can cause the resulting values to be exceptionally large and using integers longer than 64 bits can't be avoided in practice. Number of bits (including sign bit) required for representation of results of used operations with respect to range of input point coordinates is shown in Table 5.

Table 4. Numerical ranges of results in used projective space operations for three-dimensional geometry.

Object	Equation	Coordinates	Range
Input point ($\mathbf{P}$)	$\mathbf{P} = \left[x, y, z, 1\right]$	x, y, z	$\left[-n, n\right]$
		w	1
Normal of a hyperplane ($\mathbf{N}$)	$\mathbf{N} = \mathbf{P}_1 \times \mathbf{P}_2 \times \mathbf{P}_3$	x, y, z	$\left[-6n^2, 6n^2\right]$
		w	$\left[-6n^3, 6n^3\right]$
Intersection point ($\mathbf{Q}$)	$\mathbf{Q} = \mathbf{N}_1 \times \mathbf{N}_2 \times \mathbf{N}_3$	x, y, z	$\left[-1296n^7, 1296n^7\right]$

		w	$\left[-1296n^6, 1296n^6\right]$
Input point check versus line	$\mathbf{N} \cdot \mathbf{P}$		$\left[-24n^3, 24n^3\right]$
Intersection point check versus line	$\mathbf{N} \cdot \mathbf{Q}$		$\left[-31104n^9, 31104n^9\right]$

Table 5. Number of bits (including sign bit) required for representation of results of used operations with respect to range of input point coordinates.

Range	10^3	10^4	10^5	10^6	10^7	10^8	10^9
n	11	15	18	21	25	28	31
$6n^2$	24	31	37	44	51	57	64
$6n^3$	34	44	54	64	74	84	94
$24n^3$	36	46	56	66	76	86	96
$1296n^6$	72	92	111	131	151	171	191
$1296n^7$	82	105	128	151	175	198	221
$31104n^9$	106	136	166	196	226	256	286

Example Application of Operations in RP^3

To prove usefulness of geometric operations in $\mathbf{RP}^3$ space a simple algorithm performing Boolean operations on solids will be presented. For the purpose of this algorithm, a solid is be represented by its boundary defined using polygonal mesh. Every polygon is defined by a loop of edges and a directed plane, which this polygon lies on. This plane is positioned in such way, that its positive side faces interior of the solid. Every edge consists of a pair of its vertices and a plane used to define the line running along this edge. The intersection of this plane and the plane associated with the face results in a line running along the edge. Moreover, the plane associated with the edge is positioned in such way, that the polygon lies on its positive side. We assume that every mesh being considered is closed and correctly defines the solid. We also assume that no polygon has any three of its vertices collinear. To perform CSG operations on polygon meshes we should first define algorithms performing more basic operations on polygons, so we can proceed to define the algorithm performing required CSG operations.

Cutting polygon with a plane

Being given a polygon and a directed plane we can use this plane to split polygon in two. This operation can be accomplished with following algorithm:

1. For each polygon vertex determine the side of the cutting plane the vertex lies on or whether this vertex lies on this plane.
2. If there are no vertices on the positive side of the cutting plane or if there are no vertices on its negative side, stop, as the plane doesn't intersect the polygon.
3. Split each edge which starting and ending points lie on opposite sides of the cutting plane. Two edges are created in place of the edge being split. The middle point of the split is the intersection of the cutting plane, the plane associated with the edge and the plane associated with the polygon.
4. Create first of resulting polygons from edges lying on the positive side of the cutting plane. Close this polygon by adding an edge with vertices lying on the cutting plane (if initial polygon was correct, there are exactly two such vertices). Use cutting plane as the plane associated with this edge.
5. Create second of resulting polygons from edges lying on the negative side of the cutting plane. Close this polygon by adding an edge with vertices lying on the cutting plane. Use cutting plane with its sides swapped as the plane associated with this edge (to accomplish this reverse the normal of hyperplane defining the cutting plane in $\mathbf{RP}^3$ space).

Checking polygon intersection

The second operation on polygons we define is operation checking whether two polygons touch or intersect. Two polygons are defined to be touching if their intersection is a segment of non-zero length and one of the polygons lies completely on one side of the other one. Two polygons are defined to be intersecting if their intersection is a segment of non-zero length and both polygons can be split into two parts of non-zero surface using the plane associated with the other polygon. If none of the above is true, we define polygons as not intersecting. Cases of single vertex of one polygon touching the other are to be treated as cases of not intersecting polygons. The following algorithm can be used to determine whether polygons touch, intersect or don't intersect at all.

1. Test whether bounding boxes of polygons are touching for fast detection of most cases when polygons don't touch nor intersect.

2. Check all vertices of the first polygon against the plane associated with the surface of the second polygon. If all vertices of the first polygon lie on the positive side of this plane or if all these vertices lie on the negative side of it, stop, as polygons don't touch nor intersect. If all these vertices lie on this plane, also stop, as this means that polygons are coplanar and different algorithm should be used to handle this case.

3. Check all vertices of the second polygon against the plane associated with the surface of the first polygon. If all vertices of the second polygon lie on the positive side of this plane or if all these vertices lie on the negative side of it, stop, as polygons don't touch nor intersect.

4. Use information gathered in step 2. to find intersection points of the boundary of the first polygon and the plane associated with the second polygon. Vertices lying on this plane are such intersection points. If any edge of the first polygon has one of its vertices lying on the positive side and the other lying on the negative side of this plane, this edge intersects the plane and this

intersection point can be computed as intersection of planes associated with both polygons and the plane associated with the edge.

5. There should be exactly two intersection points found in step 4. If only a single vertex lies on the plane, stop, as polygons don't touch nor intersect. The case when there are more than two intersection points can happen only if we allow three or more collinear vertices to be present in polygons, which we didn't.

6. Create a segment running along the line of intersection of planes associated with the polygons. Intersection points found in step 4. are the ends of this segment.

7. Clip the segment with each plane associated with edges of the second polygon to find what part of the segment that lies on positive side of all these planes. To clip the segment with a plane check which sides of the plane the ends of the segment lie on. If neither end of the segment lies on the negative side of the plane proceed to the next plane. If one of the ends lies on the negative side of the plane and the other one doesn't lie on its positive side, the segment is completely culled away and initial polygons are determined to be not touching nor intersecting. If none of the above is true, one end of the segment lies on the positive side and the other one lies on the negative side of the plane, so the segment has to be clipped. The end point lying on the negative side has to be replaced with intersection point of the plane being considered and the line running along the segment, which line is defined as intersection of planes of both polygons.

8. If during step 7. the segment wasn't culled away, the polygons touch or intersect. To determine which is true use information gathered in steps 2. and 3. If at least one of the polygons has all its vertices lying on one side or on the plane associated with the other polygon, the polygons are determined to be touching. Otherwise, the polygons are determined to be intersecting.

Determining whether polygon lies inside a solid

Being given a polygon, we can check whether it lies inside a solid described using a polygonal mesh. We assume here that polygon doesn't intersect with any polygons of the mesh (it still can touch them) and that no part of the polygon lies on the surface of the solid represented by the mesh. The idea behind this algorithm is to shoot a ray from one of vertices of the polygon being checked and find nearest polygon of the mesh to determine which side of this polygon the polygon being tested lies on. The following algorithm can be used to perform this task:

1. Create a segment representing the ray being shot. Pick one of edges of the polygon being tested and use line associated with this edge as the line running along the segment. Use one of vertices of this edge as the starting point of this segment.

2. To find ending point of the segment we can find intersection of the line running along the segment with workspace boundary. A boundary plane of the workspace can be obtained by creating a plane running through three corners of the workspace lying on one of its sides, for example $[n, n, n]$, $[-n, n, n]$ and $[n, -n, n]$, where n is maximum coordinate value allowed for a point from input data. Any plane being boundary of the workspace can be picked as long as the line intersects it. If it's not the case, different boundary plane should be used.

3. Clip the end of the segment with all polygons of the mesh defining the solid. For each of these polygons check which sides of the plane associated with this polygon the points of the segment lie on.

4. If both points of the segment lie on positive side or both lie on the negative side of the polygon, skip this polygon as the segment doesn't intersect with it and return to step 3. to check next polygon.

5. If both points of the segment lie on the polygon, skip this polygon and return to step 3. to check next polygon, as the segment is coplanar with the polygon being skipped, so this polygon is of no use when determining the result of the algorithm.

6. Find intersection point of the line associated with the segment and the plane associated with the polygon. Check whether the intersection point lies inside the polygon. This point lies inside the polygon if and only if it doesn't lie on negative side of any plane associated with edges of this polygon. If intersection point doesn't lie inside the polygon, skip this polygon as the segment doesn't really intersect with it and return to step 3. to check next polygon.

7. The segment intersects the polygon, so replace ending point of the segment with found intersection point. Remember the polygon used to find this intersection point and return to step 3. to check remaining polygons.

8. When all polygons were checked, the polygon remembered most recently in step 7. is the polygon hit by the ray. If no polygon was remembered during this step, the ray didn't intersect the mesh and the initial polygon is determined to be lying outside the solid.

9. Check all vertices of the initial polygon being tested against the plane associated with the polygon remembered most recently in step 7. If any vertex of the initial polygon being tested lies on the positive side of this plane, the polygon is determined to be lying inside the solid. If any vertex of this polygon lies on the negative side of the plane, the polygon is determined to be lying outside the solid. If both or none of the above is true, it means that one of the assumptions made for this test wasn't true.

Finding common part of surface of two polygons

Being given a pair of coplanar polygons we can find common part of their surface. To do this, we have to split each polygon using planes associated with edges of the other polygon. A part of any of these polygons that lies on positive sides of all planes associated with edges of the other polygon is the common surface of the polygons. Following algorithm can be used to perform this operation:

1. Split the first polygon using planes associated with edges of the second polygon. After each split, the part lying on the negative side of the cutting plane is determined to be a part of the first polygon lying outside the second polygon. The part lying on the positive side should be split further using planes associated with remaining edges of the second polygon. If there is no part of the polygon lying on the positive side, stop, as the polygons don't intersect.

2. Perform operations described in step 1. with roles of the polygons switched to find parts of the second polygon lying inside and outside the first polygon.

After the above algorithm finishes, each polygon is partitioned into parts lying outside and inside the other polygon.

Simple CSG algorithm for meshes

Having defined the above algorithms we can proceed to define the algorithm performing CSG operations on polygonal meshes. The algorithm presented here is slightly modified version of algorithm described in [Laidlaw86]. Being given two polygonal meshes, $\mathbf{A}$ and $\mathbf{B}$, we have to found which parts of these meshes lie inside and outside the other mesh. To do this, we first have to make sure that no two polygons of these meshes intersect and split any intersecting polygons if needed, so the intersection of their surfaces is removed. If any two polygons from meshes $\mathbf{A}$ and $\mathbf{B}$ share a part of their surface, such polygons should also be split in such way, that their common part is a single polygon and their remaining parts don't touch the other polygon. After removing all intersections each resulting polygon should be determined to be OUTSIDE the other mesh or INSIDE it. In case of polygons sharing their surface, they are determined to be facing in the SAME or OPPOSITE directions. Following algorithm accomplishes this task:

1. For each pair of polygons from meshes $\mathbf{A}$ and $\mathbf{B}$ determine, whether they touch or intersect using one of previously presented algorithms. If they only touch or don't intersect at all, proceed to the next pair of polygons.

2. If the polygons intersect, split each of these polygons using the plane associated with the other polygon then return to step 1. and consider next pair of polygons. Note that new polygons were created during this step and they also should be considered during step 1.

3. If the algorithm used in step 1. fails because polygons are coplanar, extract polygon being common part of their surface using one of the algorithms presented earlier. In effect, each of these polygons is partitioned into polygons lying entirely outside the other polygon and a polygon sharing entire surface with the other polygon. Mark resulting polygons determined to be sharing their surface as SAME or OPPOSITE according to direction of normals of planes associated with these polygons. To check whether these planes are facing the same direction, use dot product on normals of hyperplanes representing these planes, ignoring the w component of these normals (as it can be seen in Equation 5 first three components of hyperplane normal are the normal of the real plane). Polygons are determined to be SAME if the result of the dot product is positive. Otherwise, they are OPPOSITE.

4. Return to step 1. until all pairs of polygons were considered.

5. After each pair was considered, for each resulting polygon that wasn't determined to be SAME or OPPOSITE, determine whether it lies INSIDE or OUTSIDE of the other solid using one of the algorithms presented earlier.

After classifying all the polygons and splitting them when needed we can choose which polygons will be included in the resulting mesh according to Boolean operation being performed, as shown below.

$$\mathbf{A} \cup \mathbf{B} = \mathbf{A}_{OUTSIDE} \cup \mathbf{B}_{OUTSIDE} \cup \mathbf{A}_{SAME}$$
$$\mathbf{A} \cap \mathbf{B} = \mathbf{A}_{INSIDE} \cup \mathbf{B}_{INSIDE} \cup \mathbf{A}_{SAME}$$
$$\mathbf{A} \setminus \mathbf{B} = \mathbf{A}_{OUTSIDE} \cup r(\mathbf{B}_{INSIDE}) \cup \mathbf{A}_{OPPOSITE}$$

Operation $r(*)$ is used to reverse facing direction of all polygons in $\mathbf{B}_{INSIDE}$, which can be done by changing sign of all components of hyperplane normals representing planes associated with polygons being reversed. As all algorithms presented here are based entirely on integer operations in $\mathbf{RP}^3$ space, they it can be proven to work with all possible sets of valid input data. Because this algorithm generates T-joints in resulting meshes, it may not be useable in all applications. To address this problem the algorithm should be extended to work on mesh data with connectivity information, but this extension is outside the scope of this chapter and will not be presented here as this algorithm was only demonstrated as a proof-of-concept for integer operations using $\mathbf{RP}^3$ space. Also many optimizations, like using more advanced spatial structures to speed up finding intersecting polygons can be used to improve efficiency of the algorithm presented.

References

[Laidlaw86] Laidlaw, David H., et al, “Constructive Geometry for Polyhedral Objects”, Computer Graphics, Vol. 20, no. 4 (SIGGRAPH 1986): pp. 161-170