Design, Verification and Implementation of a Polygon Clipping Application Using Co-Design Techniques

by
Thomas S. Hall & Kenneth B. Kent

TR07-182, June 14, 2007

Faculty of Computer Science
University of New Brunswick
Fredericton, N.B. E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
Email: fcs@unb.ca
www: http://www.cs.unb.ca

Abstract

This paper describes the process of designing and verifying a hardware/software co-designed system. This is done by going through a complete case study involving polygon clipping algorithms as applied to computer graphics. As is the case in many software and hardware/software design processes, verification of the software part of the system is done using test scenarios while the hardware partition is verified using the SystemC Verification Standard methodology. This case study carries the design process through to a partial integration of the hardware and software partitions using SystemC simulation.

1. Introduction

Clipping is a technique used to bound the size of an image or shape in graphics [1,2]. This paper is limited to the application of hardware/software co-design techniques and verification of polygon clipping algorithms; an example of image clipping can be found in [3].

The development of the hardware/software co-design for the Weiler-Atherton algorithm for polygon clipping was developed in three major steps. First a software-only design of the algorithm and supporting input and output operations was created using normal object-oriented design techniques [6, 7]. During this design process, consideration was given to the future partitioning of the design into hardware and software components. Two practical dividing points were identified and defined as the boundaries between classes in the object-oriented design. The object-oriented design was implemented in C++ and tested.

The second stage of the development process consisted of determining if viable communications could be established between the hardware and software components of the system. For this to be determined, the software-only design was modified so that one part of the system ran as software on a standard host computer while the other part was converted to execute as embedded software on the Altera DE2 development board [11]. One of the previously identified possible hardware/software partitioning points was used to divide the software application.

The third stage of the design process was to create the design for the hardware partition of the polygon clipping system. This was done by selecting the same hardware/software division point used in the communications test and converting the portion that had been previously used as embedded software into the hardware partition design. The hardware partition design was then implemented and verified using SystemC [8–10].

Section 2 defines the terminology and mathematical formulae used in polygon clipping. Section 3 describes some of the polygon clipping algorithms that have been developed by researchers. Section 4 begins the design process for a hardware/software co-designed implementation of the Weiler-Atherton polygon clipping algorithm used in the remainder of the paper [1]. Section 5 describes the design of the hardware partition of the system. Section 6 describes the implementation and verification process used. Finally, Section 7 provides some conclusions drawn from this work and proposes some future work based on it. The Appendices contain the final versions of the source code for the three test implementations and simulations created during the work.

2. Definitions

This section provides definitions for some of the terminology used in the remainder of the paper along with equations for computing the specific values for some of the terms defined. The definitions given here are sufficient for the purposes of this work and are not intended to be mathematically rigorous.

1. *Point* - A location in space specified in some coordinate system. In this work, the 2-dimension coordinate system is used with x representing distance along the horizontal axis and y representing distance along the vertical axis. A point p is specified as

$p = (x, y)$ where $0 \leq x \leq 254$ and $0 \leq y \leq 254$.

2. *Line Segment* - A line segment is a portion of a line passing through two points on a plane. The segment has endpoints at the two specified points that describe it.

3. *Line Segment Intersection* - Two line segments are said to intersect if the two lines of which they are part intersect and that intersection occurs within the bounds of the two line segments. The point of intersection of two line segments can be found using the following computations where two line segments have the endpoints a and b, and c and d respectively (See [1] pages 131 - 135).

$$D = (b_x - a_x)(d_y - c_y) - (b_y - a_y)(d_x - c_x) \tag{1}$$

Equation 1 determines if the two line segments are parallel or not and a value exists for t_0 in Equation 2. If D is non-zero then the two line segments are not parallel. Parallel segments are indicated by a value of zero for D, a special case in polygon clipping that is not dealt with in this paper.

$$t_0 = \frac{(c_x - a_x)(d_y - c_y) - (c_y - a_y)(d_x - c_x)}{D} \tag{2}$$

$$u_0 = \frac{a_x + t_0(b_x - a_x)}{c_x + t_0(d_x - c_x)} = \frac{a_y + t_0(b_y - a_y)}{c_y + t_0(d_y - c_y)} \tag{3}$$

Equations 2 and 3 together determine if an intersection of the two line segments exists. An intersection exists if both $0 \leq t_0 \leq 1$ and $0 \leq u_0 \leq 1$. There are two methods for computing u_0 in order to avoid divide-by-zero issues if horizontal or vertical line segments are involved in a computation.

$$i_x = a_x + t_0(b_x - a_x), i_y = a_y + t_0(b_y - a_y) \tag{4}$$

Once it is determined that an intersection point exists, the x and y coordinates of the intersection are found using Equation 4.

4. *Polygon* - A polygon is a closed region on the plane (using the two-dimensional coordinate system) formed by placing line segments on the plane such that each line segment shares each of its endpoints with one other line segment. For any given line segment in the polygon, its endpoints must be shared with two separate line segments. Further, no two line segments can intersect or touch other than at their endpoints (See [4, 5] for a formal definition). The polygon clipping algorithms described in the next section place an ordering on the points forming line segments. This ordering is necessary for the determination of the interior and exterior of a polygon by these algorithms.

5. *Convex Polygon* - For the purposes of this work a convex polygon is defined as a polygon with all interior angles less than 180 degrees.

6. *Inside-Outside Test* - An ordering can be placed upon the points that comprise the endpoints of the line segments that make up a polygon. When determining how another line segment intersects one the line segments of a polygon when such an ordering is applied it is necessary to determine which of the endpoints of the intersecting line is inside the polygon's line segment and which is outside. When the ordering placed upon the polygon's points is such that the interior of the polygon is to the right of all of its line segments, then for some point $p = (p_x, p_y)$ and line segment ab where $a = (a_x, a_y)$ and $b = (b_x, b_y)$, the inside-outside test is given in Equation 5. See [1] pages 243 - 245.

$$d = \vec{n} \cdot \vec{t} = (n_x * t_x) + (n_y * t_y) \tag{5}$$

where d is the dot product of $\vec{n}$ and $\vec{t}$.
$\vec{n}$ is the normal vector to the line segment ab passing through the origin. It can be represented by the point $n = (n_x, n_y)$ where $n_x = -(b_y - a_y)$ and $n_y = b_x - a_x$.
$\vec{t}$ is the difference of the vectors $\vec{a}$ and $\vec{p}$ formed with the origin and points a and p respectively. $t_x = p_x - a_x$ and $t_y = p_y - a_y$.

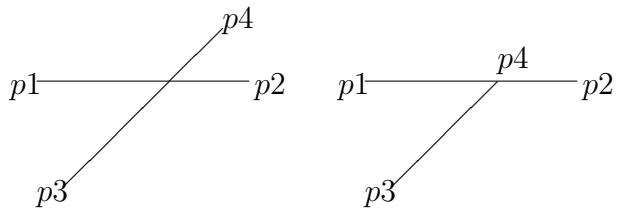
If $d < 0$ then p is inside ab, else if $d > 0$ then p is outside ab, otherwise p lies on ab [1].

3. Polygon Clipping Algorithms

The purpose of polygon clipping is to determine what portion of a given polygon lies within some bounding window. By the definition of a polygon given in Section 2(1), a polygon is made up of line segments. Thus the function of a polygon clipping algorithm is to determine what line segment (or parts thereof) from the given polygon fall within the bounds of the window. Assuming that the window itself is also defined as a polygon, this becomes, largely, a problem of line segment intersection. Therefore, prior to describing any polygon clipping algorithms, a discussion of line segment intersection is necessary.

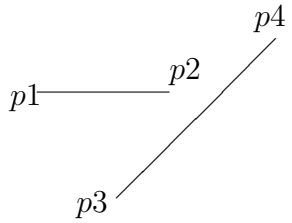
Figure 1 shows some of the possible ways in which two line segments can or cannot intersect. There are two factors to consider when determining whether two line segments intersect; are the endpoints of one line segment left, right or on the other line segment, and do the segments actually intersect. In order to determine if a point is to the left, right or on a line segment an ordering must be applied to the end points of the line segment. Assume that the line segments are ordered such that $p1$ precedes $p2$ and $p3$ precedes $p4$.

A given point is to the left of a line segment if when travelling from the starting point of a line segment to its ending point the given point is to the left. In Figure 1(a) and (c) point $p3$ is to the right of segment $(p1, p2)$ while point $p4$ is to the left. In part (b) of the figure point $p3$ is to the right of the segment $(p1, p2)$ and $p4$ is on the line segment ($p1$, $p4$ and $p2$ are co-linear).Figure 1(d) shows both points $p3$ and $p4$ to the right of the segment $(p1, p2)$.

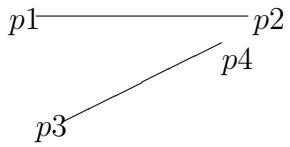


(a) Intersecting line segments (no co-linearity)

(b) Intersecting line segments (with co-linearity)



(c) Non-intersecting line segments



(d) Non-intersecting line segments

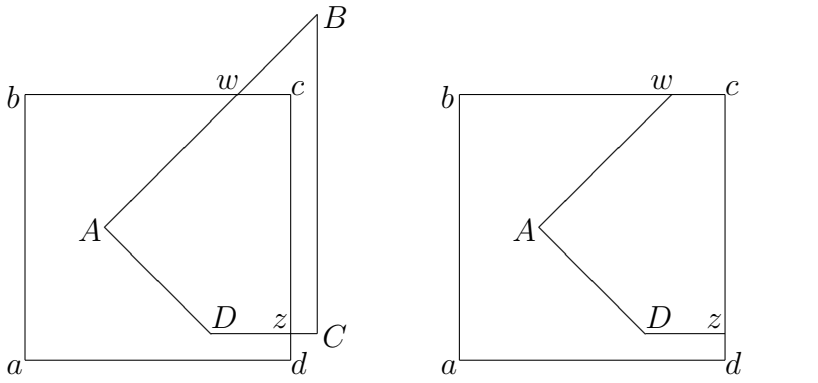
Figure 1. Line segment intersection examples.

An inspection of Figure 1 shows that it is easy to determine whether a given point is left or right of a line segment visually, but this is not the case when the representation is a list of points. In that case the use of the formulae in Section 2 for determining if a point is inside or outside a polygon must be used. These formulae are the same for both line segments and polygons, only their application is different. See [1] for more information.

The determination of whether the end points of a line segment are left or right of another line segment are not sufficient to determine if the line segments intersect. Figure 1 (a) and (c) show the cases where $p3$ is to the right of segment $(p1, p2)$ and $p4$ is to the left. In (a) the segments $(p1, p2)$ and $(p3, p4)$ intersect while in (c) they do not. When point $p3$ and $p4$ are on the same side of the line segments, as in (d), then the segments will not intersect. These cases are covered by the line segment intersection formulae given in Section 2. There are a number of special cases when determining line segment intersection centering around co-linearity of the points defining the end points of the line segments. An example of this is shown in Figure 1 (b) where $p4$ lies on the segment $(p1, p2)$. These special cases are discussed in detail in [1] and are left for future work.

Figure 2 shows a simple example of polygon clipping. This example will be used in the discussion of clipping algorithms below. In (a) the window is shown as the rectangle *abcd* and the subject polygon as the quadrilateral $ABCD$. The intersection points of the two shapes are points w and z. In part (b) of the figure, the window is *abcd* as in part (a) but the polygon is now $AwczD$. An observer seeing only (b) could also say that the polygon is $DzdabwA$. The proper interpretation of the clipping operation is a key function of any polygon clipping algorithm.

Hill [1] describes two polygon clipping algorithms, the Sutherland-Hodgman and Weiler-



(a) Polygon and Window (b) Clipped Polygon and Window

Figure 2. A simple example of polygon clipping.

Atherton algorithms. These two algorithms are described in this section. The Weiler-Atherton algorithm is the basis for the work described in the remainder of this paper.

3.1. Sutherland-Hodgman Algorithm

The Sutherland-Hodgman algorithm utilizes a convex window against which to clip an arbitrarily shaped subject polygon. This permits the algorithm to take advantage of the special characteristics of convex polygons and the line segments that form them, in particular, halfspaces [1, 4, 5]. Following the conventions defined in Section 2, consider the directed line segment ab of the window polygon in Figure 2 and the infinite line of which it is part. Let the inside halfspace of this line be to the right of the segment ab and the outside halfspace be to the left. All of the points forming the subject polygon $ABCD$ lie within the inside halfspace and are retained at this point. See Figure 3(a).

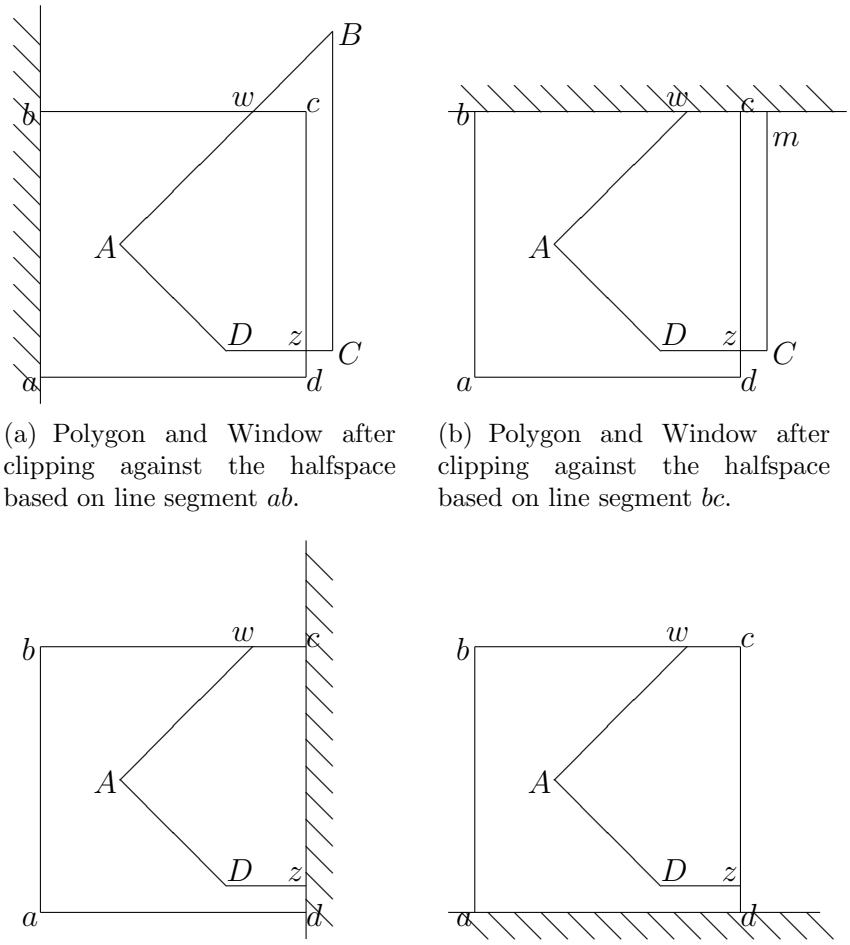
In Figure 3(b), clipping is done using the halfspace defined by the line through points b and c. Point B lies in the outside halfspace for this line. This results in two line segments being clipped, AB becomes Aw and BC becomes mC. Repeating the process for the halfspace along the line passing through points c and d, the segment mC lies totally outside the halfspace and segment CD is clipped at point z as shown in (c). The halfspace on the line through points d and a does not result in any clipping action. The resulting polygon is $AwczD$.

A drawback of this algorithm is that when there are multiple polygons in the result there will be extra line segments defined between adjacent clipping points in different polygons. This can cause problems with other applications using the output from this algorithm [1].

3.2. Weiler-Atherton Algorithm

The Weiler-Atherton polygon clipping algorithm overcomes the limitations of the Sutherland-Hodgman algorithm. It supports both non convex subject and window polygons. It also avoids the addition of extra edges to the results of the clipping operation [1].

The input to the Weiler-Atherton polygon consists of two sets of arbitrary polygons. One set defines the clipping window and the other is the clipping subject. The window set must contain at least one polygon that defines the outer bound of the clipping window. Any



(a) Polygon and Window after clipping against the halfspace based on line segment *ab*.

(b) Polygon and Window after clipping against the halfspace based on line segment *bc*.

(c) Polygon and Window after clipping against the halfspace based on line segment *cd*.

(d) Polygon and Window after clipping against the halfspace based on line segment *da*.

Figure 3. Sutherland-Hodgman algorithm applied to the example in Figure 2.

additional polygons in this set are nested inside the outer boundary polygon and may be further nested within each other. The subject polygons set can consist of any number of polygons, some of which may be nested inside other polygons. Any nested polygon in either set must be wholly contained within the polygon in which it is nested. The ordering of the points in any polygon must be such that the inside of a polygon lies to the right of any adjacent pair of points in the ordered lists of points [1].

Two trivial cases exist in the application of this algorithm; the subject polygon lies entirely within the window, and the subject polygon lies entirely outside the window. In the first case, the subject is included in its entirety in the output. In the second case, if the window lies inside the subject polygon then image clipping may be required to determine what is output, else there is no output since the subject and window polygons share no common part of the plane on which they lie. The third case is the non-trivial one where line segments of the window and subject polygons intersect. The remainder of this paper deals with this non-trivial case.

The Weiler-Atherton algorithm begins by proceeding one line segment at a time around the subject polygon until a line segment is found that intersects the window polygon where the starting point of the line segment of the subject polygon lies outside the window and the ending point of the same line segment lies inside the window. The intersection point is computed. A new polygon is created in the output and this newly computed point is added to it. In addition the new point is added to the subject and window polygons such that in each polygon two new line segments are created each consisting of one of the points in the original intersecting segments and the new point. The ordering of the points in these new segments must maintain the ordering of points in the entire polygon. In Figure 2, the window is *abcd* and the subject polygon is $ABCD$. Following the defined ordering, the first intersection between the two polygons where the beginning point of the line segment is on the outside and the ending point inside occurs when the subject polygon is at CD and the window segment is *cd*. The intersection point is z so the subject polygon becomes $ABCzD$ and the window *abczd*. A new output polygon is created and z is added to it. Processing continues adding all ending points of line segments from the subject polygon that are within the window until an ending point outside the window is found. Thus, the next two points added to the output for the polygons in Figure 2 are D and A respectively producing the output zDA. At this time another intersection point is computed between segments AB and *bc*. The resulting point is w. This new point is added to the output to give $zDAw$ and the subject and window polygons become $AwBCzD$ and *abwczd* respectively.

Since the line segment of the subject polygon that produced point w was leaving the polygon it is known that for the time being at least no further intersections will occur between the segments of the polygons. The window and subject polygons are swapped and processing continues. Point c lies inside the polygons $AwBCzD$ that is now the window so it is added to the output producing an output of $zDAwc$. At the next step the subject line segment is cz but z is already in the output indicating that the end of an output polygon has been reached. In this example this is the complete output since there is only one resulting polygon. When there are multiple inside to outside and outside to inside line segment crossings, the above process is repeated as necessary to find all of the clipped polygons.

Special cases of intersections exist, in particular when line segments are parallel or when the endpoint of one line segment is co-linear with the endpoints of a segment in the other polygon. These special cases are not described here and their implementation in hardware is left for future work.

4. Application Design

This section describes the initial software only design of the polygon clipping algorithm. In the next section a detailed description of the hardware/software co-designed version is given. The transition between the two designs is part of the verification process described in Section 6.

Figure 4 shows the class diagram for the software only version of the application using standard Unified Modelling Language (UML) notation [6, 7]. The attributes and operations (functions) of each class are described in the tables following the class diagram in order to keep the diagram simple. Before considering the details of each class, a brief description of

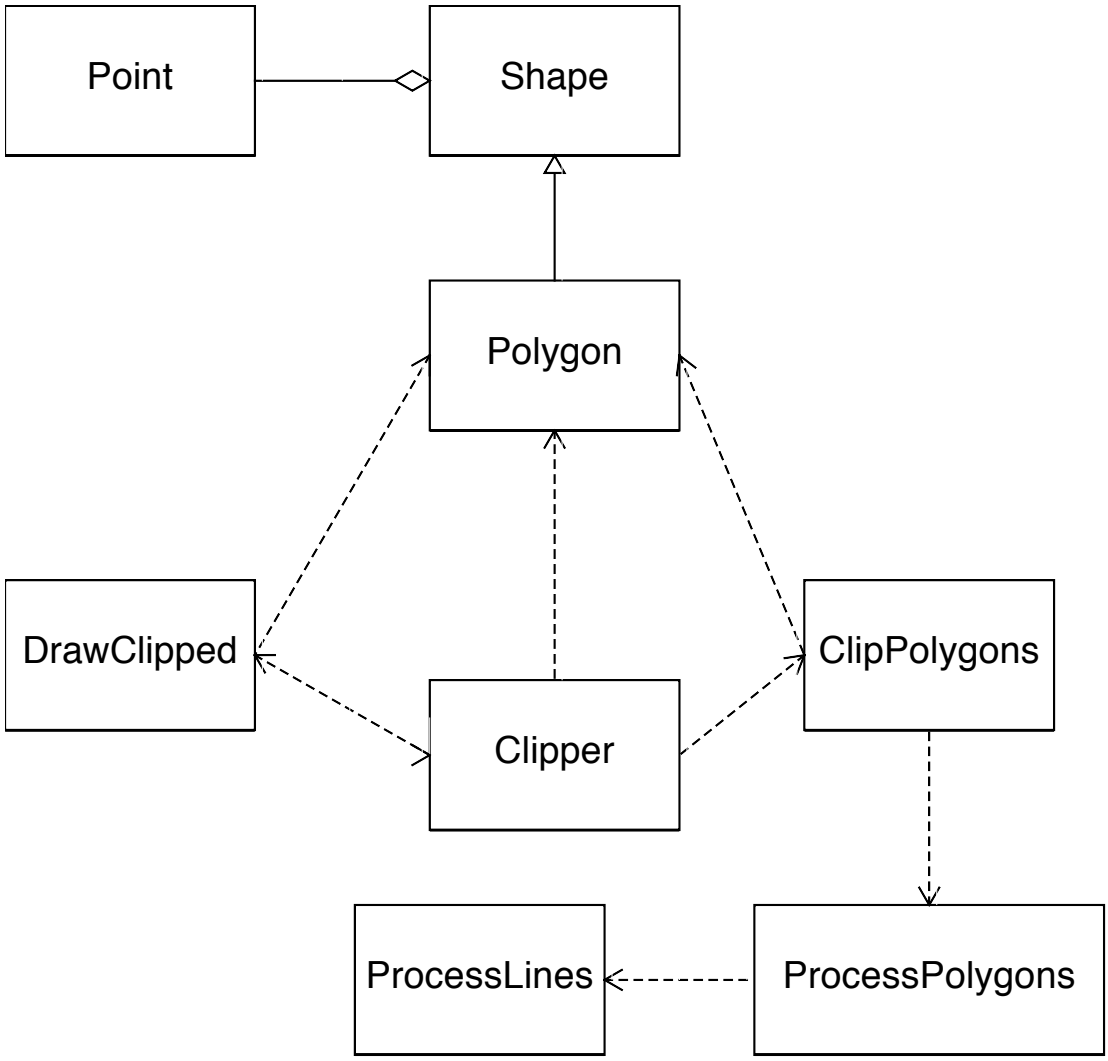


Figure 4. Class diagram of the software only implementation of the polygon clipping application.

each and the relationship between them is appropriate.

The class **Point** provides a representation for a single location on a plane. It includes the attributes and operations necessary to create, access and maintain such a representation. The ***Shape*** class is an abstract representation of a geometric shape represented by an ordered set of points as described in Section 2. Class **Polygon** is a concrete instance of ***Shape*** that describes a polygon as defined in Section 2. These three classes together provide the attributes and operations necessary to represent a polygon containing holes as in the Weiler-Atherton polygon clipping algorithm in an object oriented manner.

The **ClipPolygons**, **ProcessPolygons** and **ProcessLines** classes provide the functionality to perform polygon clipping. **ClipPolygons** provides the interface for the clipping algorithm that is implemented in **ProcessPolygons** while **ProcessLines** implements the intersection determination operations described in Section 2. Only the **ProcessPolygons**

Point
int x int y
Point(int pX, int pY) int getX() int getY()

Figure 5. Class Point attributes and operations.

class has any knowledge of the clipping algorithm actually being used. In an implementation where only software is to be used, classes **ClipPolygons** and **ProcessPolygons** could be combined. The choice to separate them here permits two possible hardware to software transition points: between these two classes and between **ProcessPolygons** and **ProcessLines**. In this work only the latter option is described. No provision for supporting polygons with nested polygons is included in these classes.

DrawClipped provides a means for outputting the results of the clipping operation. The output format is implementation dependent. Finally, the **Clipper** class provides overall control of the application and the functionality to load polygon data.

In the following subsections each of the classes discussed above is described in more detail. No attempt is made in these descriptions to define attributes and operations of these classes as private, protected or public. In most cases they are defined as public, although, in some cases they have been changed to private during implementation. The choice to define everything as public initially was intended to grant the SystemC verification tools access to all items. This was found to be unnecessary in some cases and the definition of many operations and some attributes were changed during implementation and verification. The source listings in the Appendices contain the final versions of the class definitions.

4.1. Class Point Description

The class **Point** provides a representation for a single point on a two-dimensional plane. This representation is used in the construction of geometric shapes in that plane. The structure of the class is shown in Figure 5.

Point contains two integer attributes x and y. These are the coordinates of the point along the horizontal and vertical axes respectively. There are three operations within the **Point** class:

- Point(int pX, int pY) - This is a class constructor. The two integer parameters are used to set the corresponding coordinate attributes of the instance of the class created when the constructor is called.
- int getX() - This operation returns the value stored in the horizontal (x) coordinate attribute of a specified class instance.
- int getY() - This operation returns the value stored in the vertical (y) coordinate attribute of a specified class instance.

Shape
Point **points int pointPosition int pointCount int numberPoints
void addPoint(Point *p) Point *getPoint() Point *getPointPrior() int getPointCount() int getPointPosition() Point *getPointAt(int psn) Point *getPointPriorAt(int psn) Point *getPointNextAt(int psn) void pointPositionReset() void initShape(int numPoints) Shape() ∼Shape()

Figure 6. Class Shape attributes and operations.

4.2. Class *Shape* Description

Class ***Shape*** provides an abstract representation of an arbitrary geometric figure in a two-dimensional plane. The attributes and operations of this class are generalized to allow any shape that can be described as a sequence of points to be represented by a subclass of ***Shape***. Figure 6 shows the structure of this class.

The attributes for class ***Shape*** are:

- Point **points - A reference to a list of points. This type of representation is used in order to allow the number of points representing a geometric shape to be determined at runtime.
- int pointPosition - The index of the current point in the list.
- int pointCount - The actual number of points in the list.
- int numberPoints - The maximum possible number of points in the list.

Class ***Shape*** has the following operations:

- void addPoint(Point *p) - This operation adds an instance of the **Point** class to a concrete instance of a subclass of the ***Shape*** abstract class. The point is added as the last point in the ordered set of points.

- Point *getPoint() - This operation returns the **Point** object instance stored at the current position within the list of points. The current position is determined by the value of attribute pointPosition described above. The value of pointPosition is advanced to the next position after the current point is retrieved with wrap around occurring at the end of the list.

- Point *getPointPrior() - This operation returns the point immediately prior to the current point in the ordered list. The pointPosition attribute is not changed during this operation, thereby maintaining the current position. Wrap around occurs if point-Position is at the beginning of the list.

- int getPointCount() - This operation returns the number of **Point** object instances in the list (i.e. the value of pointCount).

- int getPointPosition() - This operation returns the index of the current **Point** object instance (i.e. pointPosition).

- Point *getPointAt(int psn)- This operation returns the **Point** object instance at the specified position (psn) in the list.

- Point *getPointPriorAt(int psn) - This operation returns the **Point** object instance that precedes the one at the specified position (psn). If the specified position is the first in the list, the last entry is returned.

- Point *getPointNextAt(int psn) - This operation returns the **Point** object instance that follows the one at the specified position (psn). If the specified position is the last in the list, the first entry is returned.

- void pointPositionReset() - This operation sets the pointPosition attribute to the first entry in the list.

- void initShape(int numPoints) - This operation is intended for use by subclasses of the ***Shape*** abstract class to initialize the data elements defined in ***Shape***. The parameter numPoints is the number of points that will be used to represent the geometric shape defined by the concrete subclass.

- Shape() - A do-nothing constructor.

- ∼Shape() - This operation is a destructor for the ***Shape*** abstract class. It cleans up the data items defined within the abstract class when instances of concrete subclasses are destroyed.

4.3. Class Polygon Description

The class **Polygon** is a concrete subclass of the class ***Shape***. A polygon is thus defined as an ordered set of points where each adjacent pair of points represents the endpoints of a line segment in the polygon. The last and first points together also represent a line segment

Polygon
Polygon **nestedPolygons int nestedCount int nestedPosition int numberNested
void addNested(Polygon *nested) Polygon *getNested() Polygon *getNestedAt(int psn) int getNestedCount() int getNestedPosition() void nestedPositionReset() Polygon(int numPoints, int numNested) ∼ Polygon()

Figure 7. Class Polygon attributes and operations.

to produce a closed figure. In addition, many applications have polygons that have other polygons nested within them. The nesting of polygons is supported by this class definition by including references to one or more other polygons within a polygon. Figure 7 shows the structure of this class.

The attributes for class **Polygon** are:

- Polygon **nestedPolygons - A list of instances of class **Polygon** that are nested within this instance of the class. A dynamic structure is used for this list to enable runtime determination of the number of nested polygons, if there are any.
- int nestedCount - The number of nested polygons in the list. There may be zero or more polygons in the list.
- int nestedPosition - The index of the current polygon in the list of nested polygons.
- int numberNested - The total number of polygons that can be in the list of nested polygons.

The operations for class **Polygon** are:

- void addNested(Polygon *nested)- This operation adds an instance of class **Polygons** (as parameter nested) to another instance of the class. Care must be taken to ensure that an instance is not passed a reference (pointer) to itself as a nested polygon.
- Polygon *getNested() - This operation retrieves the current nested polygon from within the one being worked with as determined by the value of the nestedPosition attribute. The current nested polygon is advanced to the next one in the list at the end of this operation with wrap-around at the end of the list.
- Polygon *getNestedAt(int psn) - This operation retrieves the nested polygon at the specified position (psn) within the list of nested polygons. The nestedPosition is not changed by this operation.

DrawClipped
void draw(Polygon *poly)

Figure 8. Class DrawClipped attributes and operations.

- int getNestedCount() - This operation returns the number of nested polygons in the list (i.e. the value of nestedCount).
- int getNestedPosition() - This operation returns the index of the current position in the list of nested polygons (i.e. the value of nestedPosition).
- void nestedPositionReset() - This operation sets the value of the nestedPosition attribute to the index of the first entry in the list.
- Polygon(int numPoints, int numNested) - This is a constructor for the **Polygon** class. It takes the number of points (numPoints) forming the polygon itself and the number of nested polygons (numNested) as parameters.
- $\sim$ Polygon() - Class destructor used to cleanup the data structures used when an instance of this class is no longer required.

A special case use of the **Polygon** class is storage of polygons created during a clipping operation. In this case there are no points stored directly in the top-level polygon object returned by the clipping operation. The resulting clipped polygon or polygons are stored as nested polygons within the top-level polygon object returned.

4.4. Class DrawClipped Description

As shown in Figure 8 the **DrawClipped** class is a very simple one. This version of the class is intended to output each point in a polygon in a textual list. In a graphical implementation this class would need to be revised to support that type of user interface and its relationship with the **Clipper** class would also need to be changed.

The single operation (draw) in the **DrawClipped** class accepts a polygon (poly) containing nested polygons that are the output of a clipping operation and lists the x and y coordinates of each point in those polygons in text format with labels separating the polygons.

4.5. Class Clipper Description

The class **Clipper** is the central controlling unit in the polygon clipping system. The primary function of this class is to control the flow of the system as clipping proceeds. It also provides the functionality to load polygon and window data from the data files specified by the user. Figure 9 shows the structure of the class.

The attributes for the **Clipper** class are:

Clipper
Polygon **polygons int polygonCount int polygonPosition int numberPolygons Polygon *window Polygon *clippedPolygon ClipPolygons *clipPolygons DrawClipped *drawClipped
int runClipper(char *fileName) Clipper() ∼ Clipper() int loadPolygons(char *fileName) int readAllPolygons(FILE *in_file, int numPolys) int readOnePolygon(FILE *in_file, int numPoints, int numNested, Polygon *poly) int readPoints(FILE *in_file, int numberPoints, Polygon *poly)

Figure 9. Class Clipper attributes and operations.

- Polygon **polygons - A dynamically created list of polygons, possibly with nested polygons, to be clipped.
- int polygonCount - The number of polygons to be clipped.
- int polygonPosition - The current polygon in the list.
- int numberPolygons - The maximum number of top-level polygons that can be placed in the list.
- Polygon *window - A polygon containing the window against which all other polygons are to be clipped. There can be nested polygons within the window.
- Polygon *clippedPolygon - A polygon containing nested polygons that are the result of the clipping operation. There will be no points defining a top level polygon in this polygon instance.
- ClipPolygons *clipPolygons - Reference to an instance of the **ClipPolygons** class that controls the actual clipping operation.
- DrawClipped *drawClipped - Reference to an instance of the **DrawClipped** class used for outputting the results of the clipping operation.

The operations for the **Clipper** class are:

- int runClipper(char *fileName) - This operation is the main controlling method of the entire clipping operation. It accepts the name of the polygon and window data files (fileName) as a parameter. This is just the base file name without any extensions. A

ClipPolygons
ProcessPolygons*processPolygon
Polygon *clip(Polygon *window, Polygon *poly) ClipPolygons() ∼ ClipPolygons()

Figure 10. Class ClipPolygons attributes and operations.

full path may be specified if the program is executed in a directory other than that containing the data files. The return value is an integer indicating the success or failure of reading the polygon and window data files. Any non-zero value indicates a successful load operation while a zero signifies failure.

- Clipper() - Constructor.
- ∼ Clipper() - Destructor to clean up memory when an instance of this class is no longer needed.
- int loadPolygons(char *fileName) - This operation controls the loading of the polygon and window data files. It takes the base name (possibly including a path) as input parameter fileName and returns a non-zero value of the operation succeeded or zero if it failed. Upon successful completion, the polygons and window class attributes contain valid data.
- int readAllPolygons(FILE *in_file, int numPolys) - This operation controls the reading of all the polygons contained within one data file. It takes a reference to the file to read (in_file) and the number of polygons in that file as input parameters. It returns a non-zero value on success or a zero upon failure.
- int readOnePolygon(FILE *in_file, int numPoints, int numNested, Polygon *poly) - This operation loads a single polygon and any polygons nested within it from the specified file (in_file) into the specified polygon data structure (poly). It will load numPoints points and numNested nested polygons. This operation returns a non-zero value upon success and zero on failure.
- int readPoints(FILE *in_file, int numberPoints, Polygon *poly) - This operation reads the specified number of points (numPoints) of a single polygon from the specified file (in_file) into polygon poly. The operation returns a non-zero value upon success and zero on failure.

4.6. Class ClipPolygons Description

Class **ClipPolygons** provides the functionality to clip polygons against the given window. It uses the services of the **ProcessPolygons** class to perform the actual clipping operation on any specific polygon. Figure 10 shows the structure of the class.

The **ClipPolygons** class contains one attribute:

- ProcessPolygons*processPolygon - A reference to an instance of the **ProcessPolygons** class used by this class to clip a single polygon against the given window.

The operations of the **ClipPolygons** class are:

- Polygon *clip(Polygon *window, Polygon *poly) - This operation controls the clipping of polygons (possibly nested) passed in parameter poly against the given window given by the parameter window. The resulting set of polygons is returned to the caller as nested polygons within the returned polygons.
- ClipPolygons() - This is a constructor for class instances. It creates the necessary data structures and object references.
- $\sim$ ClipPolygons() - Class instance destructor that destroys all data structures and object references used by an instance of the class.

4.7. Class ProcessPolygons Description

Class **ProcessPolygons** clips a single polygon against the given window. Clipping is done by checking each line segment within the polygon against each line segment of the window. The services of the **ProcessLines** class are used to check each pair of line segments. Figure 11 shows the structure of the class.

The design of this class is not necessarily the most efficient from a software-only implementation perspective. This was done as this class could be implemented in the hardware partition of the system design in a later version of the system. The data structures it uses are also based on this concept.

The **ProcessPolygons** class contains a single attribute:

- ProcessLines *processLines - A reference to an instance of the **ProcessLines** class that provides line segment processing for an instance of this class.

The operations of the **ProcessPolygons** class are:

- int processPolygons(int *poly1, int *poly2, int count1, int count2, int *resultPoly) - This operation provides access to the functionality of the **ProcessPolygons** class. It returns the number of polygons produced when the input polygon (parameterpoly1) is clipped against the window (parameter poly2). The resulting polygons, represented as an ordered set of points, is passed back to the caller via the resultPoly parameter. The count1 and count2 parameters contain the number of points in poly1 and poly2 respectively.
- ProcessPolygons() - This constructor creates the necessary instance of the **ProcessLines** class.
- $\sim$ ProcessPolygons() - This destructor ensures that the object references used by an instance of this class are properly cleaned up when the instance is being destroyed.

ProcessPolygons
ProcessLines *processLines
int processPolygons(int *poly1, int *poly2, int count1, int count2, int *resultPoly) ProcessPolygons() ~ ProcessPolygons() void findIntersections(int *poly1, int *poly2, int *count1, int *count2, int *resultPoly, int *countR) int findEnterPoint(int *poly, int count) void swapIntegers(int *v1, int *v2) void swapPointers(int **p1, int **p2) void insertPoint(int x, int y, int *poly, int psn, int *pcount, int status) void insertResultPoint(int x, int y, int *poly, int psn,int *pcount) bool addPoint(int x, int y, int *poly, int *pcount) void setStatus(int *poly, int psn, int status) int getStatus(int *poly,int psn) void insertInsidePoints(int *poly1, int *poly2, int max1, int max2, int *resultPoly, int *maxR) int resultPolygonCount(int *poly, int count) void listPolygon(int *poly, int count) void listPolygonR(int *poly, int count)

Figure 11. Class ProcessPolygons operations and attributes.

- void findIntersections(int *poly1, int *poly2, int *count1, int *count2, int *resultPoly, int *countR) - This operation implements the actual Weiler-Atherton polygon clipping algorithm. It accepts the subject polygon and given window as parameters poly1 and poly2. The number of points in these two polygons are given by the parameters count1 and count2 respectively. The set of polygons produced by the clipping operation is returned via parameter resultPoly with the number of points in the set returned via countR. All of these parameters are pointers (except count1 and count2) in order to support runtime data structure size determination. This operation is the one that would require the most modification if the clipping algorithm were to be changed.

- int findEnterPoint(int *poly, int count) - This operation finds a point in the subject polygons where the selected point is outside the window or is on a line segment as the entry point of a line segment. This is required for correctly beginning the execution of the clipping algorithm. The parameter poly is the polygon in which to search and count is the number of points in the polygon. The return value is the index of the point at which to start algorithm execution.

- void swapIntegers(int *v1, int *v2) - The Weiler-Atherton algorithm involves the swapping of the subject polygon and window as described in Section 3. This operation is used to swap integer values such as indices and point counts. The parameters are the two integer values to be swapped.

- void swapPointers(int **p1, int **p2) - This operation swaps reference values to a subject polygon and window. The parameters are the references to swap.

- void insertPoint(int x, int y, int *poly, int psn, int *pcount, int status) - This operation inserts a point given by its x and y coordinates in parameters x and y respectively into the polygon (parameter poly).The insertion is done in point psn in the polygons with the status value passed in the parameter status. The pcount parameter inputs the current number of points in the polygon poly and upon return from the operation contains the new number of points in the polygon.

- void insertResultPoint(int x, int y, int *poly, int psn,int *pcount) - This operation inserts a new point into the result polygon passed to the operation as parameter poly. The new point is given as parameters x and y respectively. The new point is inserted at index psn in the polygon. The parameter pcount initially contains the number of points in the polygons and is set to the new size of the polygon upon exit from the operation.

- bool addPoint(int x, int y, int *poly, int *pcount) - This operation appends a new point to the end of a result polygon. The point is passed into the operation as its x and y coordinates. The pcount parameter initially contains the current number of points in the polygon and upon operation exit contains the new number of points in the polygon.

- void setStatus(int *poly, int psn, int status) - This operation sets the status value of a point in either the subject polygon or window. The status of the point at index psn is changed.

- int getStatus(int *poly,int psn) - This operation retrieves the status of the point at index psn in either the subject polygon or window.

- void insertInsidePoints(int *poly1, int *poly2, int max1, int max2, int *resultPoly, int *maxR) - This operation inserts points that form part of the subject polygon that fall within the window but do not lie on the window line segments. The parameters poly1 and poly2 are references to the subject polygon and window. The resultPoly parameter is a reference to the set of polygons produced by the clipping operation.

- int resultPolygonCount(int *poly, int count) - This operation counts and returns the number of polygons in the set of polygons produced by the clipping operation.

- void listPolygon(int *poly, int count) - Test and debugging operation for listing the contents of the subject polygon and window data structures as passed to it by parameter poly. Parameter count is the number of points in the polygon being output.

- void listPolygonR(int *poly, int count) - This operation is similar to listPolygon except that it is used for outputting result polygons.

ProcessLines
int newX1 int newY1
int processLinePair(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y, int p4X, int p4Y) int getNewX() int getNewY() int checkInside(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y) float computeT0(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y, int p4X, int p4Y) float computeU0(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y, int p4X, int p4Y, float t0) int checkIntersect(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y, int p4X, int p4Y, float t0, float u0)

Figure 12. Class ProcessLines operations and attributes.

4.8. Class ProcessLines Description

The **ProcessLines** class provides the functionality to determine if two line segments, each defined by a pair of points, intersect. In addition to determining if an intersection exists, the direction of the intersection is also determined based upon the ordering of the pairs of points that make up each line segment.

In the verification process described in the Section 6 this class forms the hardware partition. It is also the portion of the design that is implemented as the software running on the embedded processor in the communications testing prototype.

The **ProcessLines** class contains two attributes:

- int newX1 - The x coordinate of an intersection point between two line segments determined by the operations in this class. The 'no-point found' value is 255. This value is reset at the beginning of processing for each pair of line segments.
- int newY1 - The y coordinate of an intersection point between two line segments determined by the operations in this class. The 'no-point found' value is 255. This value is reset at the beginning of processing for each pair of line segments.

The operations of the **ProcessLines** class are:

- int processLinePair(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y, int p4X, int p4Y) - This operation is the external interface for the class. It accepts the two line segments as pairs of points. Points p1 and p2 define one line segment while points p3 and p4 define the other. Each point is input as its x and y coordinates as the parameters of this operation. The return value is the direction of intersection (if there is one). The return values can be: ADD_NONE (0) when no intersection occurs. ADD_NEW_POINT_ENTER (1) is returned when the new point to be added is produced as a result of p1 being outside the line segment formed by points p3 and p4

and p2 inside it. ADD_NEW_POINT_LEAVE (2) is returned when the new point to be added is produced as a result of p2 being outside the line segment formed by points p3 and p4 and p1 inside it.

- int getNewX() - This operation returns the current value of attribute newX1.
- int getNewY() - This operation returns the current value of attribute newY1.
- int checkInside(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y) - This operation determines if a point (given by its x and y coordinates as parameters p3X and p3Y respectively) is inside or outside a line segment as given by equation 5. The line segment is defined by its endpoints defined by parameters p1X, p1Y, p2X and p2Y. It returns one of the constant values: INSIDE, OUTSIDE or ON_LINE indicating the result of the operation.
- float computeT0(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y, int p4X, int p4Y) - This operation computes the $t0$ intermediate value used in the equations given in Section 2. It accepts the two line segments under consideration at a given time defined by their end-points (parameters p1X, p1Y, p2X and p2Y define one line segment while p3X, p3Y, p4X and p4Y define the other). The returned value of $t0$ is a floating point value.
- float computeU0(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y, int p4X, int p4Y, float t0) - This operation computes the $u0$ intermediate value used in the equations given in Section 2. It accepts the two line segments under consideration at a given time defined by their end-points (parameters p1X, p1Y, p2X and p2Y define one line segment while p3X, p3Y, p4X and p4Y define the other). The intermediate value $t0$ is also passed to this operation. The returned value of $u0$ is a floating point value.
- int checkIntersect(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y, int p4X, int p4Y, float t0, float u0) - This operation determines if two line segments intersect. It returns one of the constant values INTERSECT or NO_INTERSECT. The inputs to the operation are the two line segments being compared as p1X, p1Y, p2X and p2Y; and p3X, p3Y, p4X, p4Y respectively. It also uses the intermediate values $t0$ and $u0$ in its computations.

5. Hardware/Software Co-design Version

This section describes the hardware/software co-designed version of the polygon clipping application. It extends the software design given in the previous section to support the use of both hardware and software operations.

As stated previously, there are two possible points within the software-only design of the system where hardware software partitioning could be done. These points are the interface between classes **ClipPolygons** and **ProcessPolygons** or between **ProcessPolygons** and **ProcessLines**. The design described here utilizes the second option.

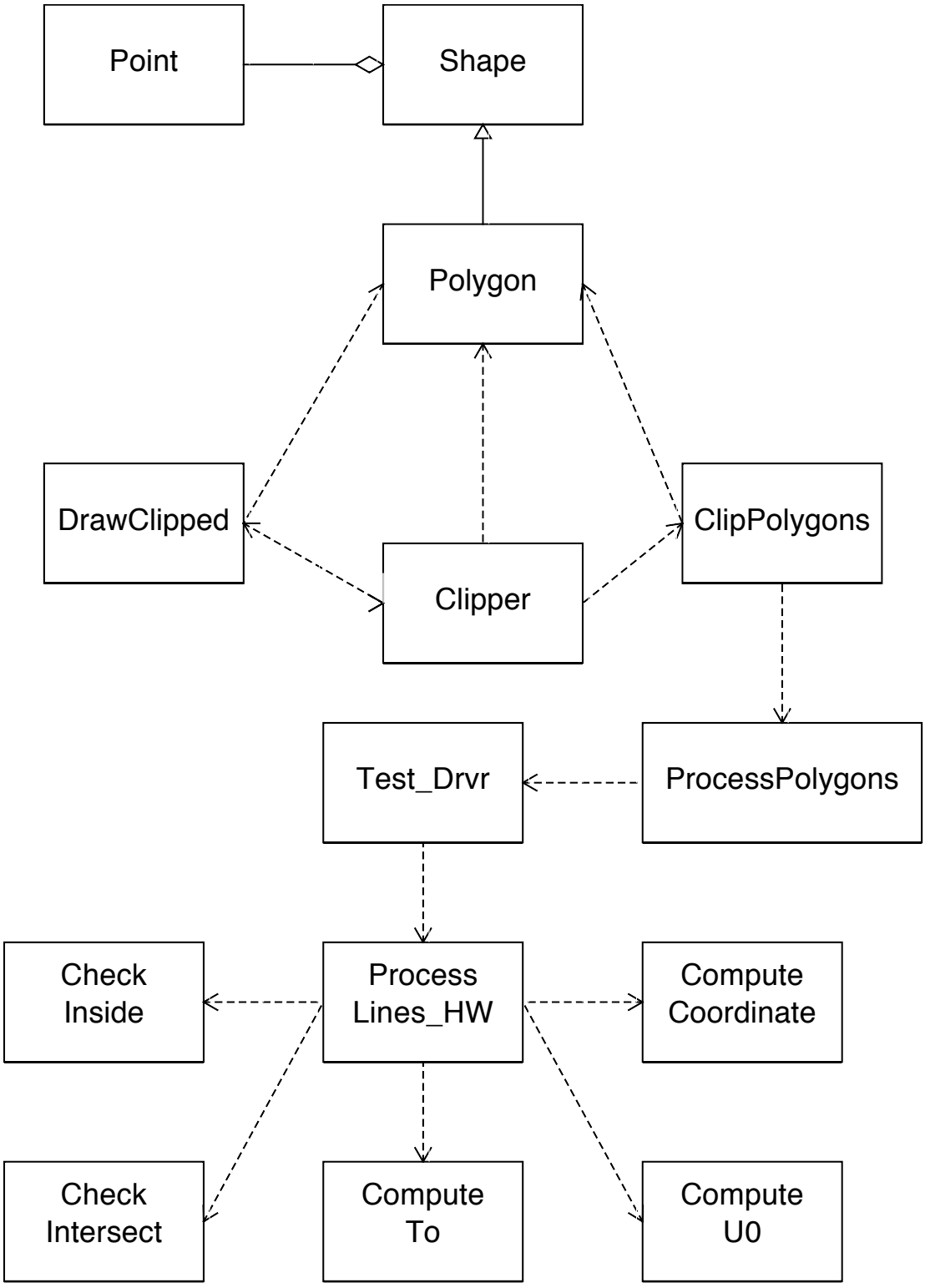


Figure 13. Class diagram of the hardware/software co-designed implementation of the polygon clipping application.

Based on this choice, all of the classes in the design except **ProcessPolygons** and **ProcessLines** are identical to those of the software-only version and are not discussed in this section. Class **ProcessPolygons** needs only slight modification to support hardware communications rather than direct invocation of operations in class **ProcessLines**. This modification is implementation dependent and has no effect on the basic design of the class, so it will not be discussed further in this section. An example of one possible implementation is given in the embedded software communication test implementation, see Appendix B.

Class **ProcessLines**, though, requires major changes. In order to properly support hardware operations this class is broken up into a number of separate classes each representing a separate hardware process. An additional class **Test_Drvr** is added to the design to simulate the communications between the hardware and software partitions. Note that in most cases the names of the new classes in this version of the design correspond to names of the operations they replace from the original **ProcessLines** class. Figure 13 shows the structure of the re-designed system.

The design is described in terms of SystemC and SystemC Verification Standard data structures and operations [8–10]. Since the SystemC class libraries make extensive use of C++ templates the actual C++ class definitions of the hardware partition classes are given here rather than UML models to make the descriptions of the classes easier to understand.

5.1. Class Test_Drvr Description

Class **Test_Drvr** provides an interface between the hardware and software partitions of the system. It executes within the software partition. Figure 14 shows the definition of the class.

The declaration of the **Test_Drvr** class begins with several constant declarations that are discussed later in this section.

The class is then defined using the SC_MODULE macro. This macro creates a class using the sc_module template to define a class that operates within the the SystemC simulation environment. Within the macro the attributes, data structures and operations of the class are defined.

The attributes of the class are:

```
sc_in_clk         clk;
sc_out<sc_bv<9> > p1Xs;
sc_out<sc_bv<9> > p1Ys;
sc_out<sc_bv<9> > p2Xs;
sc_out<sc_bv<9> > p2Ys;
sc_out<sc_bv<9> > p3Xs;
sc_out<sc_bv<9> > p3Ys;
sc_out<sc_bv<9> > p4Xs;
sc_out<sc_bv<9> > p4Ys;
sc_out<sc_logic > data_sets;
sc_in<sc_bv<2> >  result_outs;
sc_in<sc_bv<9> > resultX_outs;
sc_in<sc_bv<9> > resultY_outs;
```

```
#define STATE_WAIT_INPUT  (sc_uint<2>)1
#define STATE_HW_DISPATCH (sc_uint<2>)2
#define STATE_WAIT_OUTPUT (sc_uint<2>)3
#define CTRL_DATA_WAIT    1
#define CTRL_DATA_READY 2

SC_MODULE(Test_Drvr) {
    sc_in_clk         clk;
    sc_out<sc_bv<9> > p1Xs;
    sc_out<sc_bv<9> > p1Ys;
    sc_out<sc_bv<9> > p2Xs;
    sc_out<sc_bv<9> > p2Ys;
    sc_out<sc_bv<9> > p3Xs;
    sc_out<sc_bv<9> > p3Ys;
    sc_out<sc_bv<9> > p4Xs;
    sc_out<sc_bv<9> > p4Ys;
    sc_out<sc_logic > data_sets;
    sc_in<sc_bv<2> >  result_outs;
    sc_in<sc_bv<9> > resultX_outs;
    sc_in<sc_bv<9> > resultY_outs;
    sc_in<sc_logic >  result_sets;

    struct TPointStruct{
        sc_int<9> p1X;
        sc_int<9> p1Y;
        sc_int<9> p2X;
        sc_int<9> p2Y;
        sc_int<9> p3X;
        sc_int<9> p3Y;
        sc_int<9> p4X;
        sc_int<9> p4Y;
        sc_int<9> resultX;
        sc_int<9> resultY;
        sc_uint<2> result;};

    void runTest();
    int processLinePair(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y,
                        int p4X, int p4Y, int *newX, int *newY);
    SC_CTOR(Test_Drvr){
        SC_CTHREAD(runTest, clk.pos());}
};
```

Figure 14. Class Test_Drvr definition.

```
sc_in<sc_logic >  result_sets;
```

These attributes are all template instantiations of SystemC class library templates except the first. The sc_in_clk item defines a simulator clock input to the class used for controlling execution timing. The remaining items are definitions of communication signals between this class and others during hardware simulation. The sc_in template defines a communications path that inputs data to an instance of this class while the sc_out template defines data communications leaving a class instance. The sc_bv template defines a vector (array) of logic bits that are associated with one another and communicated as a single unit. All of the bit vectors in these definitions are 9 bits long except result_outputs which is 2 bits long. The first eight vectors represent the x and y coordinates of the endpoints of the two line segments. Points p1 and p2 correspond to one line segment, p3 and p4 represent the second segment. The result_outputs vector passes the result of processing back to the operations of this class from the process of the **ProcessLines_HW** class. The resultX_outputs and resultY_outputs signals return the new x and y value when an intersection point is found. The data_sets and result_sets signals are defined as sc_logic items. They are single bit logic items that provide control signals between the process in this class and the process of the **ProcessLines_HW** class. These two signals will be discussed in detail later in this section.

The data structure in the class has the following definition:

```
struct TPointStruct{
    sc_int<9> p1X;
    sc_int<9> p1Y;
    sc_int<9> p2X;
    sc_int<9> p2Y;
    sc_int<9> p3X;
    sc_int<9> p3Y;
    sc_int<9> p4X;
    sc_int<9> p4Y;
    sc_int<9> resultX;
    sc_int<9> resultY;
    sc_uint<2> result;};
```

The elements of this data structure are instances of SystemC templates that correspond to the communications signals defined as attributes above on a one-to-one basis with matching names (without the trailing 's'). These items are the local storage locations for the data passed via the communication signals. The control signals do not have corresponding entries in this data structure, instead they use local variables within the process.

The operations of the class are:

- void runTest() - This operation provides control of the communications between an instance of this class and an instance of the **ProcessLines_HW** class. The flow of communications is described below.

- int processLinePair(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y, int p4X, int p4Y, int *newX, int *newY) - This operation provides the software interface to the

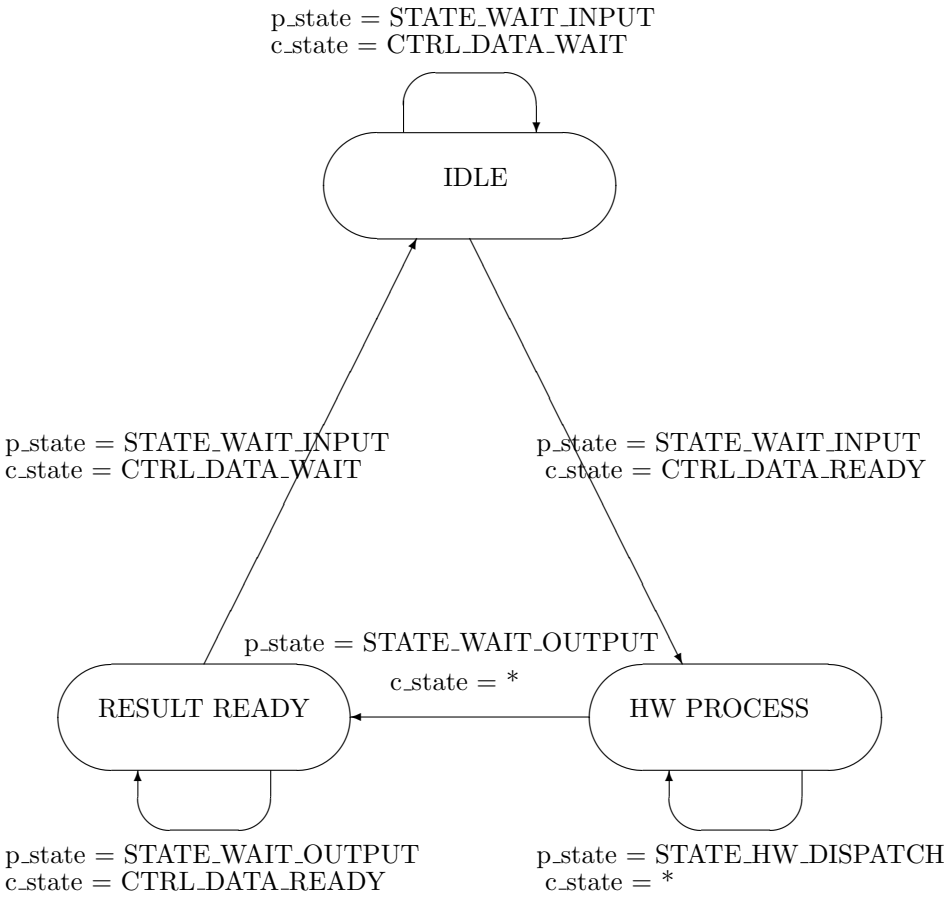


Figure 15. State diagram for the runTest process function of the Test_Drvr class.

class. The return value is the result of checking for the existence of an intersection between the two input line segments. The input parameters to the function are four points corresponding to the end-points of the two line segments being checked for intersection. The parameters newX and newY are output parameters that contain the intersection point of the two input line segments if one exists.

- SC_CTOR(Test_Drvr){ SC_CTHREAD(runTest, clk.pos());} - This is a constructor for the class created using the SC_CTOR macro. Within the constructor, the SC_THREAD macro is used to invoke the runTest operation as a process with the internal operation triggered by the positive edge of the the input clock signal clk.

The *runTest* operation in the **Test_Drvr** is defined to operate as a SystemC thread process [8]. This means that it executes as a separate thread within the SystemC simulator and its execution is sensitive to the positive edge of the simulation clock. Because of this definition, *runTest* needs to implement a state machine as shown in Figure 15. This state machine uses two state variables *p_state* for controlling the flow of hardware processing and

c_state for controlling software processing. *c_state* can have two values:

- CTRL_DATA_WAIT - The software portion of the system can supply data to the instance of the **Test_Drvr** class for passing on to the hardware partition.
- CRL_DATA_READY - The software partition has provided data for hardware processing.

The hardware management state variable *p_state* can take on one of three values:

- STATE_WAIT_INPUT - The hardware is idle waiting for data input from the software partition.
- STATE_HW_DISPATCH - Data has been received from the software portion and passed onto the hardware partition.
- STATE_WAIT_OUTPUT - Results have been received from the hardware partition and is waiting for the software partition to retrieve those results.

The values for the two state variables that cause state transitions are shown in Figure 15.

5.2. Class ProcessLines_HW

The class **ProcessLines_HW** (known as ProcessLines in the SystemC simulation code given in Appendix C) is the first hardware partition class definition. This class provides communication facilities between the hardware and software partitions on the hardware side. It also communicates with other hardware processes (classes in the SystemC simulation) that perform the actual computations in determining whether an intersection point exists between two line segments. The interface of this class is shown below.

```
#define STATE_WAIT_SET   (sc_uint<4>)1
#define STATE_INSIDE_1   (sc_uint<4>)2
#define STATE_INSIDE_2   (sc_uint<4>)3
#define STATE_T0         (sc_uint<4>)4
#define STATE_U0         (sc_uint<4>)5
#define STATE_INTERSECT  (sc_uint<4>)6
#define STATE_NEW_X      (sc_uint<4>)7
#define STATE_NEW_Y      (sc_uint<4>)8
#define STATE_SEND_DATA  (sc_uint<4>)9
#define STATE_WAIT_UNSET (sc_uint<4>)10

SC_MODULE(ProcessLines)
{
public:
// CheckInside interface
    sc_out<sc_bv<9> > ins_p1Xs;
    sc_out<sc_bv<9> > ins_p1Ys;
```

```
    sc_out<sc_bv<9> > ins_p2Xs;
    sc_out<sc_bv<9> > ins_p2Ys;
    sc_out<sc_bv<9> > ins_p3Xs;
    sc_out<sc_bv<9> > ins_p3Ys;
    sc_out<sc_logic > ins_data_sets;
    sc_in<sc_bv<2> >  ins_result_outs;
    sc_in<sc_logic >  ins_result_sets;

// CheckIntersect interface
    sc_out<sc_bv<9> > int_p1Xs;
    sc_out<sc_bv<9> > int_p1Ys;
    sc_out<sc_bv<9> > int_p2Xs;
    sc_out<sc_bv<9> > int_p2Ys;
    sc_out<sc_bv<9> > int_p3Xs;
    sc_out<sc_bv<9> > int_p3Ys;
    sc_out<sc_bv<9> > int_p4Xs;
    sc_out<sc_bv<9> > int_p4Ys;
    sc_out<float >    int_t0s;
    sc_out<float >    int_u0s;
    sc_out<sc_logic > int_data_sets;
    sc_in<sc_bv<2> >  int_result_outs;
    sc_in<sc_logic >  int_result_sets;

// ComputeT0 interface
    sc_out<sc_bv<9> > t0_p1Xs;
    sc_out<sc_bv<9> > t0_p1Ys;
    sc_out<sc_bv<9> > t0_p2Xs;
    sc_out<sc_bv<9> > t0_p2Ys;
    sc_out<sc_bv<9> > t0_p3Xs;
    sc_out<sc_bv<9> > t0_p3Ys;
    sc_out<sc_bv<9> > t0_p4Xs;
    sc_out<sc_bv<9> > t0_p4Ys;
    sc_out<sc_logic > t0_data_sets;
    sc_in<float >     t0_result_outs;
    sc_in<sc_logic >  t0_result_sets;

// ComputeU0 interface
    sc_out<sc_bv<9> > u0_p1Xs;
    sc_out<sc_bv<9> > u0_p1Ys;
    sc_out<sc_bv<9> > u0_p2Xs;
    sc_out<sc_bv<9> > u0_p2Ys;
    sc_out<sc_bv<9> > u0_p3Xs;
    sc_out<sc_bv<9> > u0_p3Ys;
    sc_out<sc_bv<9> > u0_p4Xs;
```

```
    sc_out<sc_bv<9> > u0_p4Ys;
    sc_out<float >    u0_t0s;
    sc_out<sc_logic > u0_data_sets;
    sc_in<float >     u0_result_outs;
    sc_in<sc_logic >  u0_result_sets;

// ComputeCoordinate interface
    sc_out<sc_bv<9> > cc_v1s;
    sc_out<sc_bv<9> > cc_v2s;
    sc_out<float >    cc_t0s;
    sc_out<sc_logic > cc_data_sets;
    sc_in<sc_bv<9> >  cc_result_outs;
    sc_in<sc_logic >  cc_result_sets;

// ProcessLines interface
    sc_in_clk         clk;
    sc_in<sc_bv<9> >  p1Xs;
    sc_in<sc_bv<9> >  p1Ys;
    sc_in<sc_bv<9> >  p2Xs;
    sc_in<sc_bv<9> >  p2Ys;
    sc_in<sc_bv<9> >  p3Xs;
    sc_in<sc_bv<9> >  p3Ys;
    sc_in<sc_bv<9> >  p4Xs;
    sc_in<sc_bv<9> >  p4Ys;
    sc_in<sc_logic >  data_sets;
    sc_out<sc_bv<2> > result_outs;
    sc_out<sc_bv<9> > resultX_outs;
    sc_out<sc_bv<9> > resultY_outs;
    sc_out<sc_logic > result_sets;

    void processLinePair();

    SC_CTOR(ProcessLines)
    {
        SC_CTHREAD(processLinePair, clk.pos());
    }
};
```

The structure of class **ProcessLines_HW** is very similar to that of **Test_Drvr**. The major difference is the number of signal definitions in the class. This is because the process operation of this class communicates with all of the processes that perform the actual computations via signals as well as with the **Test_Drvr** class process operation. The signals associated with each process are distinguished by the prefixes of the signal names as follows:

- *ins* - Signals that communicate with process operation of the **CheckInside** class.

- *int* - Signals that communicate with process operation of the **CheckIntersect** class.
- *t0* - Signals that communicate with process operation of the **ComputeT0** class.
- *u0* - Signals that communicate with process operation of the **ComputeU0** class.
- *cc* - Signals that communicate with process operation of the **ComputeCoordinate** class.
- *no prefix* - Signals that communicate with process operation of the **Test_Drvr** class.

The definitions of these signals are the same as described for the **ProcessLines_HW** class and are not repeated here. The single operation within this class is *processLinePair()*. It is defined as a thread process operation sensitive to the input clock signal. This operation provides algorithm flow control for the determination of intersection points in the hardware partition. It operates as a finite state machine using the state variables defined at the beginning of the class as shown in Figure 16. The values of the state variable *p_state* in Figure 16 have the following meanings:

- STATE_WAIT_SET - Waiting for data from the process of the **Test_Drvr** class.
- STATE_INSIDE_1 - Utilizing the process of the **CheckInside** class to determine if a given point is inside a polygon or not. The point is the starting point of a given line segment.
- STATE_INSIDE_2 - Utilizing the process of the **CheckInside** class to determine if a given point is inside a polygon or not. The point is the ending point of a given line segment.
- STATE_T0 - Utilizing the process of class **ComputeT0** to compute the $t0$ intermediate value.
- STATE_U0 - Utilizing the process of class **ComputeU0** to compute the $u0$ intermediate value.
- STATE_INTERSECT - Utilizing the process of class **CheckIntersect** to determine if the current line segments intersect or not.
- STATE_NEW_X - Utilizing the process of class **ComputeCoordinate** to determine the x coordinate of the intersection between the current line segments.
- STATE_NEW_Y - Utilizing the process of class **ComputeCoordinate** to determine the y coordinate of the intersection between the current line segments.
- STATE_SEND_DATA - Notify the process of the **Test_Drvr** class that results are available.
- STATE_WAIT_UNSET - Wait for the process of **Test_Drvr** to read the available results.

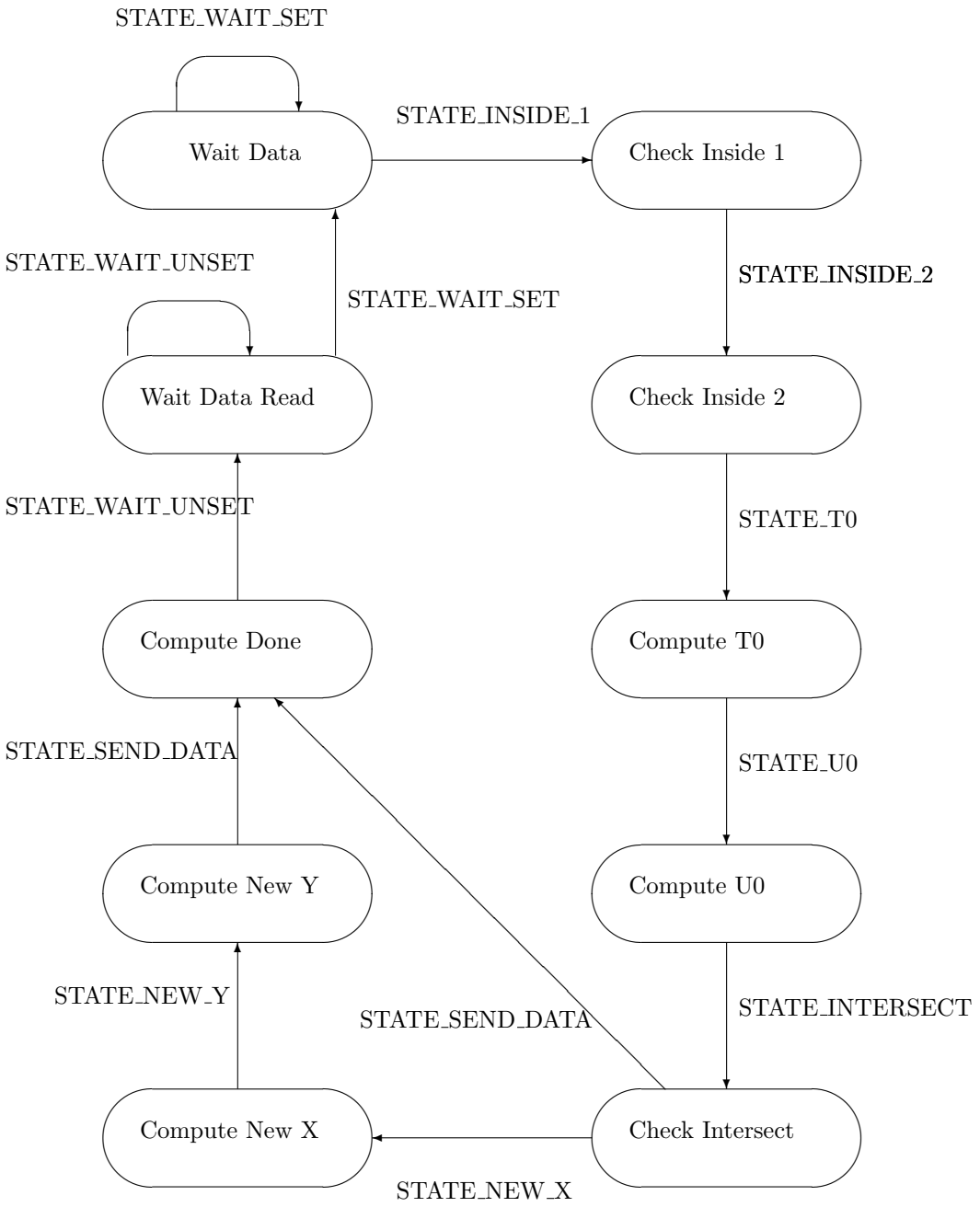


Figure 16. State diagram for the processLinePair process function of the ProcessLines_HW class. Note that the state variable p_state is implied on all transitions in this diagram.

5.3. Other Hardware Partition Classes

The remaining classes in the hardware partition and their processes all behave in a like manner when viewed from outside the class. The only difference between them is the number and types of signals used to communicate with their processes. Figure 17 shows the state diagram describing the behavior of all of these classes. Below are the C++ interfaces for these classes.

```
SC_MODULE(CheckInside) {
    sc_in_clk         clk;
    sc_in<sc_bv<9> >  p1Xs;
    sc_in<sc_bv<9> >  p1Ys;
    sc_in<sc_bv<9> >  p2Xs;
    sc_in<sc_bv<9> >  p2Ys;
    sc_in<sc_bv<9> >  p3Xs;
    sc_in<sc_bv<9> >  p3Ys;
    sc_in<sc_logic >  data_sets;
    sc_out<sc_bv<2> > result_outs;
    sc_out<sc_logic > result_sets;
    void checkInside();
    SC_CTOR(CheckInside){
        SC_CTHREAD(checkInside, clk.pos());
    }
};

SC_MODULE(ComputeT0){
  sc_in_clk         clk;
  sc_in<sc_bv<9> >  p1Xs;
  sc_in<sc_bv<9> >  p1Ys;
  sc_in<sc_bv<9> >  p2Xs;
  sc_in<sc_bv<9> >  p2Ys;
  sc_in<sc_bv<9> >  p3Xs;
  sc_in<sc_bv<9> >  p3Ys;
  sc_in<sc_bv<9> >  p4Xs;
  sc_in<sc_bv<9> >  p4Ys;
  sc_in<sc_logic >  data_sets;
  sc_out<float >    result_outs;
  sc_out<sc_logic > result_sets;
  void computeT0();
  float cvt_int_float(sc_int<32> val);
  SC_CTOR(ComputeT0){
    SC_CTHREAD(computeT0, clk.pos());
  }
};
```

```
SC_MODULE(ComputeU0){
  sc_in_clk         clk;
  sc_in<sc_bv<9> >  p1Xs;
  sc_in<sc_bv<9> >  p1Ys;
  sc_in<sc_bv<9> >  p2Xs;
  sc_in<sc_bv<9> >  p2Ys;
  sc_in<sc_bv<9> >  p3Xs;
  sc_in<sc_bv<9> >  p3Ys;
  sc_in<sc_bv<9> >  p4Xs;
  sc_in<sc_bv<9> >  p4Ys;
  sc_in<float >     t0s;
  sc_in<sc_logic >  data_sets;
  sc_out<float >    result_outs;
  sc_out<sc_logic > result_sets;
  void computeU0();
  float cvt_int_float(sc_int<9> val);
  SC_CTOR(ComputeU0){
    SC_CTHREAD(computeU0, clk.pos());
  }
};

SC_MODULE(CheckIntersect){
  sc_in_clk         clk;
  sc_in<sc_bv<9> >  p1Xs;
  sc_in<sc_bv<9> >  p1Ys;
  sc_in<sc_bv<9> >  p2Xs;
  sc_in<sc_bv<9> >  p2Ys;
  sc_in<sc_bv<9> >  p3Xs;
  sc_in<sc_bv<9> >  p3Ys;
  sc_in<sc_bv<9> >  p4Xs;
  sc_in<sc_bv<9> >  p4Ys;
  sc_in<float >     t0s;
  sc_in<float >     u0s;
  sc_in<sc_logic >  data_sets;
  sc_out<sc_bv<2> > result_outs;
  sc_out<sc_logic > result_sets;
  void checkIntersect();
  SC_CTOR(CheckIntersect){
    SC_CTHREAD(checkIntersect, clk.pos());
  }
};

SC_MODULE(ComputeCoordinate){
```

```
  sc_in_clk         clk;
  sc_in<sc_bv<9> >  v1s;
  sc_in<sc_bv<9> >  v2s;
  sc_in<float >     t0s;
  sc_in<sc_logic >  data_sets;
  sc_out<sc_bv<9> > result_outs;
  sc_out<sc_logic > result_sets;
  void computeCoordinate();
  sc_int<9> round(float val);
  float cvt_int_float(sc_int<9> val);
  SC_CTOR(ComputeCoordinate){
    SC_CTHREAD(computeCoordinate, clk.pos());
  }
};
```

Figure 17 shows that the processes of these classes have only two states, waiting for data input and processing data after input. Since the state changes occur at the same time as the value of the signal *data_set*, the copy of this signal value within the processes is used as the state variable for each.

6. Implementation and Verification

This section describes the implementation of the software-only design of the polygon clipping application, an embedded software implementation and the simulation and verification of the full hardware/software co-designed version of the application.

6.1. Software-only Implementation

In Section 4, an object oriented design for the polygon clipping application was presented. An implementation of this design, using the C++ programming language was created and tested using common software testing techniques. The source code of the final version of this implementation can be found in Appendix A.

Two significant issues were uncovered during software testing. First, the rounding techniques initially used to convert floating point numbers to integers did not function as expected. This technique was modified to that shown in the source code to provide proper rounding.

The second issue discovered is data related. While the clipping algorithm is capable of processing many irregularly shaped polygons, a ordering of points was found that produced the correct intersection points but incorrectly formed polygons. As an example, consider the two orderings of points defining the same polygon in the first two columns of Table 1.

The first column in Table 1 shows the starting point as (16,6) while the second shows the starting point as (25,15), the points appear in each column in the same order except for the starting point. The third column gives the points defining the window against which the polygons is clipped.

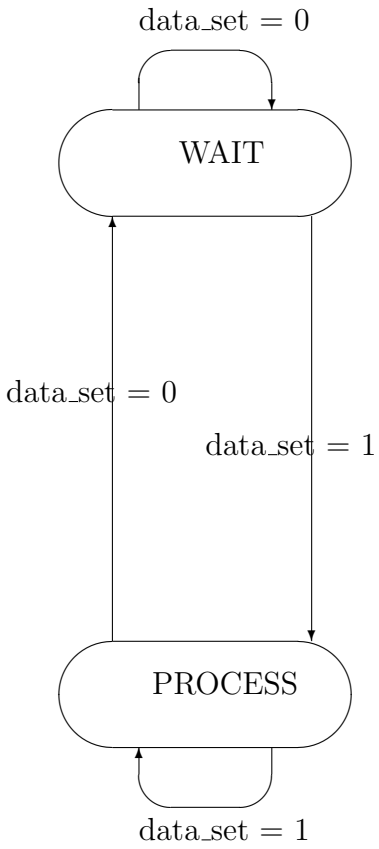


Figure 17. State diagram for the process functions of the CheckInside, CheckIntersect, ComputeT0, ComputeU0 and ComputeCoordinate classes.

Figure 18 shows a graphical view of the same data set as listed in Table 1. An examination of this view reveals that the problem with the data as shown in the first column of the table is that the top horizontal line of the window is encountered in two directions of traversal. The line segments beginning at points (16,6) and (17,8) move left to right across the top boundary of the window while the line segment beginning at (35,8) approaches right to left. This ordering of line segments results in confusion about what 'inside' the window means. By changing the order in which line segments are encountered during the clipping operation, as given by the data set in the second column of the table, this problem is avoided. This shows that the results produced by the algorithm are data dependent in its current form. The development of a data-independent algorithm is beyond the scope of this work.

Table 1. Example of how the ordering of polygon data can affect the behavior of the Weiller-Atherton algorithm.

Incorrect Point Order	Correct Point Order	Window
6,16	25,15	10,10
17,8	35,8	30,10
25,15	22,11	30,0
35,8	20,5	10,0
22,11	12,2	
20,5	6,16	
12,2	17,8	

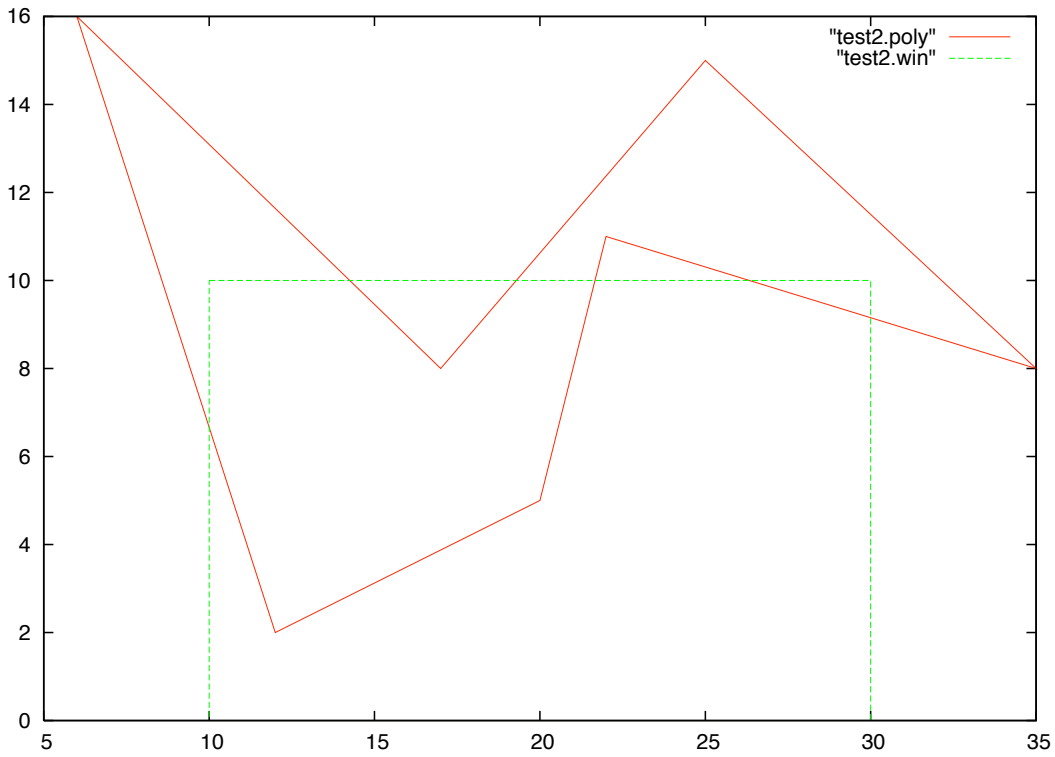


Figure 18. Graphical view of the data sets shown in Table 1.

6.2. Embedded Software Implementation

In order to test the interface between the hardware and software partitions of the system, the **ProcessLines** C++ class of the software only design was converted to a ANSI Standard C module and combined with a sample communications module designed to execute as embedded software on the FPGA on the DE2 development board from Altera [11, 12]. The **ProcessPolygons** class was also modified to include calls to the matching Dynamic Link Library (DLL) for communications in the software partition [11, 12]. The source code for the ProcessLines module and **ProcessPolygons** class used in this test appear in Appendix B.

The tests showed that it was possible to successfully communicate over the USB link between the host PC and DE2 development board. The communications time is estimated to be 16.3 milliseconds per communication (combined send and receive time for eight bytes of data). This value estimate is based on an average of 100 test communications where the number of host system clock cycles between the commencement of transmission and the completion of receipt were measured for each communication operations and averaged. For the host system used one clock cycle has a duration of 0.4167 nanoseconds and the average number of clock cycles per communication cycle is 39163264.5. This measurement technique is described in detail in [13].

6.3. SystemC Simulation and Verification

The hardware partition of the polygon clipping application has been converted to a SystemC implementation and then verified using the SystemC verification tools [8–10]. The definitions of the classes in the final version of the verification implementation are given in Section 5.

The first stage of verification creates a main program and test driver module that interface directly with the software-only version of the **ProcessLines** class. This simulation provides a baseline for comparing further simulation results against. No refinement of the **ProcessLines** class was done for this simulation.

The second through fourth stages of simulation gradually refined the **ProcessLines** class from the software-only version into a set of hardware processes. During the refinement several issues with the system were identified and resolved, some arising from coding errors in the original software version. The final version of both the software-only and hardware/software source code shown in Appendix A, B and C show the corrected code. The major problems encountered were:

- A coding error in the original software version that produced incorrect results under certain floating point to integer rounding scenarios.
- Further rounding problems were uncovered when the rounding operation was moved to a separate C++ function. This was due to the way in which C++ promotes (automatic type casting) values during arithmetic operations, explicit type casting was introduced to eliminate this problem.
- While the SystemC randomization operation provides predictable and repeatable data, it is application configuration dependant. As the refinement of the hardware partition

proceeded, the random values changed as the structure of the test main program was adjusted to suit newer versions of the simulation code.

- The fixed point numbers defined by SystemC do not appear to work correctly in the reference implementation of systemC [8].
- There is an unresolved timing issue in the final version of the simulation. This is in the interface between the hardware and software partitions and is manifested in the software partition reading the results from the hardware partition two or more times. This timing issue should disappear when proper hardware/software communications are implemented instead of the simple simulation approach used during verification. This claim is supported by the successful testing of communications using embedded software as described above.

7. Conclusion

The polygon clipping system described in this work was chosen as a case study to show the process of designing a hardware/software co-designed system and verifying its functionality. By starting with a software-only implementation of the design and gradually refining the design to a full hardware/software co-designed system permitted the testing and verification of the design at each stage of refinement. The design and simulated hardware/software co-design implementation have been shown to be valid and a full hardware partition implementation is possible.

Future work based on this work could include a full hardware partition implementation and the enhancement of the basic design to support nested polygons. A graphical user interface is another possible enhancement.

References

[1] F. S. Hill. *Computer Graphics.* Prentice Hall, 1990.

[2] D. Rogers and J. A. Adams. *Mathematical Elements of Computer Graphics, Second Edition.* McGraw-Hill In.c., 1976.

[3] M. Edwards and B. Fozard. Rapid prototyping of mixed hardware and software systems. In *Proceedings of the Euromicro Symposium on Digital System Design (DSD02)*, pages 118 – 125, Sept.4 - 6 2002.

[4] O'Rourke, J. *Computational Geometry in C.* Cambridge University Press, 1993.

[5] M. de Berg, M. Van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry, Algorithms and Applications, Second Edition.* Springer, 2000.

[6] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, Second Edition.* Addison-Wesley, 2003.

[7] E. Baude. *Software Design, From Programming to Architecture.* John Wiley and Sons Inc., 2004.

[8] Open SystemC Initiative. *SystemC 2.0.1 Language Reference Manual Revision 1.0*, 2003. Cited 10 Jan 2006, Available at www.systemc.org.

[9] Open SystemC Initiative. *SystemC Version 2.0 Users Guide Update for SystemC 2.0.1*, 2002. Cited 10 Jan 2006, Available at www.systemc.org.

[10] Open SystemC Initiative. *SystemC Verification Standard Specification Version 1.0e*, 2002. Cited 10 Jan 2006, Available at www.systemc.org.

[11] Altera Corporation. DE2 Development and Education Board, User Manual, 2005. Provided in electronic form with the board.

[12] Philips Semiconductors Corporation. An10008-01 isp1362 embedded programming guide rev: 0.9, 2002. provided in electronic form with the DE2 board.

[13] T. S. Hall. Thread level parallel execution for hardware/software co-designed virtual machines, 2005. Masters Thesis, University of New Brunswick.

A. Software Implementation

This Appendix contains the full source code of the software-only version of the polygon clipping application.

Note that the original code framework was generated using a UML toolkit. Some of the automatically generated comments remain in the code in this Appendix and those that follow.

```
#ifndef __CLIPCONSTANTS_H
#define __CLIPCONSTANTS_H

#define NO_INTERSECT 0
#define INTERSECT    1

#define INSIDE  1
#define OUTSIDE 2
#define ON_LINE 3

#define ADD_NONE            0
#define ADD_NEW_POINT_ENTER 1
#define ADD_NEW_POINT_LEAVE 2

#define NOT_USED        0
#define INTERSECT_ENTER 1
#define INTERSECT_LEAVE 2
#define INTERSECT_DONE  3

#endif
#include <iostream>
#include "Clipper.h"




int main(int argc, char **argv)
{
Clipper *clipper;

if (argc != 2)
{
std::cout << "Usage: ./clipper <data_file_name>\n";
exit(1);
}

clipper = new Clipper;

clipper->runClipper(argv[1]);

delete clipper;

return 0;
```

```
}



// C++ header file "Clipper.h" for class Clipper generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#ifndef Clipper_I12aa452cm10a55c27eb3mm7e73_H
#define Clipper_I12aa452cm10a55c27eb3mm7e73_H

#include "Polygon.h"
#include "ClipPolygons.h"
#include "DrawClipped.h"

class Clipper
{
public:
////////////////////////////////////////////////////
// public attributes
////////////////////////////////////////////////////

Polygon **polygons;
int polygonCount;
int polygonPosition;
int numberPolygons;

Polygon *window;

Polygon *clippedPolygon;


////////////////////////////////////////////////////
// public associations
////////////////////////////////////////////////////


ClipPolygons *clipPolygons;

DrawClipped *drawClipped;


////////////////////////////////////////////////////
// public operations
////////////////////////////////////////////////////

int runClipper(char *fileName);

Clipper();

~Clipper();
```

```
private:
////////////////////////////////////////////////
// private operations
////////////////////////////////////////////////

int loadPolygons(char *fileName);
int readAllPolygons(FILE *in_file, int numPolys);
int readOnePolygon(FILE *in_file, int numPoints, int numNested, Polygon *poly);
int readPoints(FILE *in_file, int numberPoints, Polygon *poly);

};

#endif // Clipper_I12aa452cm10a55c27eb3mm7e73_H
```

```
// C++ implementation file "Clipper.cpp" for class Clipper generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#include <iostream>
#include <string.h>
#include "Clipper.h"

int Clipper::runClipper(char *fileName)
{
int result;
result = loadPolygons(fileName);
if (result)
{
clippedPolygon = clipPolygons->clip(polygons[0], window);
drawClipped->draw(window);
drawClipped->draw(polygons[0]);
drawClipped->draw(clippedPolygon);
}
return result;
}

int Clipper::loadPolygons(char *fileName)
{
FILE *in_file;
char in_line[51];
char *tok;
int pts, nsts;
char winFileName[80], polyFileName[80];

winFileName[0] = '\0';
polyFileName[0] = '\0';
int result = 1;
```

```
strcpy(winFileName, fileName);
strcpy(polyFileName, fileName);
strcat(winFileName, ".win");
strcat(polyFileName, ".poly");

if ((in_file = fopen(winFileName, "r")) == NULL)
{
std::cout << "File " << winFileName << " not found\n";
return 0;
}

if (fgets(in_line, 50, in_file) != NULL)
{
tok = strtok(in_line, " ,\n");
pts = atoi(tok);
tok = strtok(NULL, " ,\n");
nsts = atoi(tok);
window = new Polygon(pts, nsts);
result = readOnePolygon(in_file, pts, nsts, window);
}

fclose(in_file);

if ((in_file = fopen(polyFileName, "r")) == NULL)
{
std::cout << "File %s " <<polyFileName << " not found\n";
return 0;
}

if (result && fgets(in_line, 50, in_file) != NULL)
{
tok = strtok(in_line, " ,\n");
pts = atoi(tok);
polygonCount = pts;
polygons = new Polygon *[pts];
result = readAllPolygons(in_file, pts);
}
else
result = 0;

fclose(in_file);

return result;;
}

int Clipper::readOnePolygon(FILE *in_file, int numPoints, int numNested, Polygon *poly)
{
int result = 1;
int i;
char in_line[51];
char * tok;
int pts;
Polygon *p_temp;
```

```
result = readPoints(in_file, numPoints, poly);

for (i = 0; result && i < numNested; i++)
{
if (fgets(in_line, 50, in_file) != NULL)
{
tok = strtok(in_line, " ,\n");
pts = atoi(tok);
p_temp = new Polygon(pts, 0);
poly->addNested(p_temp);
result = readPoints(in_file, pts, p_temp);
}
else
result = 0;
}
return result;
}

int Clipper::readAllPolygons(FILE *in_file, int numPolys)
{
int i;
int p_count, n_count;
int result = 1;
char *tok;
char in_line[51];
Polygon *p_temp;

for (i = 0; result && i < numPolys; i++)
{
if (fgets(in_line, 50, in_file) != NULL)
{
tok = strtok(in_line, " ,\n");
p_count = atoi(tok);
tok = strtok(NULL, " ,\n");
n_count = atoi(tok);
p_temp = new Polygon(p_count, n_count);
polygons[i] = p_temp;
result = readOnePolygon(in_file, p_count, n_count,p_temp);
}
else
result = 0;
}
return result;
}

int Clipper::readPoints(FILE *in_file, int numberPoints, Polygon *poly)
{
int i;
Point *p_temp;
int x, y;
char *tok;
char in_line[51];
```

```
int result = 1;


for (i = 0; i < numberPoints; i++)
{
if (fgets(in_line, 50, in_file) != NULL)
{
tok = strtok(in_line, " ,\n");
x = atoi(tok);
tok = strtok(NULL, " ,\n");
y = atoi(tok);
p_temp = new Point(x, y);
poly->addPoint(p_temp);
}
else
result = 0;
}
return result;
}

Clipper::Clipper()
{
clipPolygons = new ClipPolygons();
drawClipped = new DrawClipped();
}

Clipper::~Clipper()
{
int i;

delete window;
delete clippedPolygon;
delete drawClipped;
delete clipPolygons;

for (i = 0; i < polygonCount; i++)
delete polygons[i];
}




// C++ header file "Point.h" for class Point generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#ifndef Point_Im84629e2m10a560450ccmm7d82_H
#define Point_Im84629e2m10a560450ccmm7d82_H

class Point
{
public:
////////////////////////////////////////////////////
```

```
// public attributes
////////////////////////////////////////////////////

int x;

int y;



////////////////////////////////////////////////////
// public operations
////////////////////////////////////////////////////

Point(int pX, int pY);

int getX();

int getY();


};

#endif // Point_Im84629e2m10a560450ccmm7d82_H




// C++ implementation file "Point.cpp" for class Point generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#include "Point.h"

Point::Point(int pX, int pY)
{
x = pX;
y = pY;
}


int Point::getX()
{
return x;
}


int Point::getY()
{
return y;
}
```

```
// C++ header file "Shape.h" for class Shape generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#ifndef Shape_Im84629e2m10a560450ccmm7dad_H
#define Shape_Im84629e2m10a560450ccmm7dad_H

#include "Point.h"

class Shape
{
public:
////////////////////////////////////////////////////
// public attributes
////////////////////////////////////////////////////

Point **points;

int pointPosition;

int pointCount;

int numberPoints;


////////////////////////////////////////////////////
// public operations
////////////////////////////////////////////////////

void addPoint(Point *p);

Point *getPoint();

Point *getPointPrior();

int getPointCount();

int getPointPosition();

Point *getPointAt(int psn);

Point *getPointPriorAt(int psn);

Point *getPointNextAt(int psn);

void pointPositionReset();

void initShape(int numPoints);

Shape();
```

```
~Shape();

};

#endif // Shape_Im84629e2m10a560450ccmm7dad_H
```

```
// C++ implementation file "Shape.cpp" for class Shape generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#include "Shape.h"

void Shape::addPoint(Point *p)
{
if (pointCount < numberPoints)
{
points[pointCount] = p;
pointCount++;
}
}


Point *Shape::getPoint()
{
Point *temp;
temp = points[pointPosition];
pointPosition = (pointPosition + 1) % pointCount;
return temp;
}


Point *Shape::getPointPrior()
{
Point *temp;
int psn;

psn = pointPosition - 2;
if (psn < 0)
psn = pointCount + psn;
temp = points[psn];
return temp;
}


int Shape::getPointCount()
{
return pointCount;
}
```

```
int Shape::getPointPosition()
{
return pointPosition;
}



Point *Shape::getPointAt(int psn)
{
int p_psn = psn;
if (psn >= pointCount)
p_psn = pointCount - psn;
return points[p_psn];
}


Point *Shape::getPointPriorAt(int psn)
{
int p_psn;
p_psn = psn - 1;
if (p_psn < 0)
p_psn = pointCount - 1;
return points[p_psn];
}


Point *Shape::getPointNextAt(int psn)
{
int p_psn;
p_psn = psn + 1;
if (p_psn >= pointCount)
p_psn = p_psn - pointCount;
return points[p_psn];
}


void Shape::pointPositionReset()
{
pointPosition = 0;
}


void Shape::initShape(int numPoints)
{
numberPoints = numPoints;
pointCount = 0;
pointPosition = 0;
points = new Point *[numberPoints];
}

Shape::Shape()
```

```
{
}

Shape::~Shape()
{
int i;

for (i = 0; i < pointCount; i++)
delete points[i];
}
```

```
#ifndef __POLYGON_H
#define __POLYGON_H

#include "Shape.h"

class Polygon : public Shape
{
public:

Polygon **nestedPolygons;
int nestedCount;
int nestedPosition;
int numberNested;

void addNested(Polygon *nested);
Polygon *getNested();
Polygon *getNestedAt(int psn);
int getNestedCount();
int getNestedPosition();
void nestedPositionReset();
Polygon(int numPoints, int numNested);
~Polygon();
};

#endif
```

```
#include "Polygon.h"

void Polygon::addNested(Polygon *nested)
{
if (nestedCount < numberNested)
{
nestedPolygons[nestedCount] = nested;
nestedCount++;
}
```

```
}

Polygon *Polygon::getNested()
{
Polygon *temp;
temp = nestedPolygons[nestedPosition];
nestedPosition = (nestedPosition + 1) % nestedCount;
return temp;
}

Polygon *Polygon::getNestedAt(int psn)
{
return nestedPolygons[psn];
}

int Polygon::getNestedCount()
{
return nestedCount;
}

int Polygon::getNestedPosition()
{
return nestedPosition;
}

void Polygon::nestedPositionReset()
{
nestedPosition = 0;
}

Polygon::Polygon(int numPoints, int numNested)
{
initShape(numPoints);
numberNested = numNested;
if (numberNested > 0)
nestedPolygons = new Polygon *[numberNested];
else
nestedPolygons = 0;
nestedCount = 0;
nestedPosition = 0;
}

Polygon::~Polygon()
{
int i;

for (i = 0; i < nestedCount; i++)
delete nestedPolygons[i];
}
```

```
// C++ header file "ClipPolygons.h" for class ClipPolygons generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#ifndef ClipPolygons_I12aa452cm10a55c27eb3mm7dc0_H
#define ClipPolygons_I12aa452cm10a55c27eb3mm7dc0_H

#include "ClipConstants.h"
#include "Point.h"
#include "Polygon.h"
#include "ProcessPolygons.h"

class ClipPolygons
{
public:
////////////////////////////////////////////////////
// public attributes
////////////////////////////////////////////////////



////////////////////////////////////////////////////
// public operations
////////////////////////////////////////////////////

Polygon *clip(Polygon *window, Polygon *poly);
ClipPolygons();
~ClipPolygons();

private:
////////////////////////////////////////////////////
// private attributes
////////////////////////////////////////////////////

ProcessPolygons *processPolygon;


};
#endif




// C++ implementation file "ClipPolygons.cpp" for class ClipPolygons generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#include "ClipPolygons.h"

Polygon *ClipPolygons::clip(Polygon *window, Polygon *poly)
{
Polygon *p_new, *p_new2;
Point *pt;
```

```
int p_count1, p_count2;
int psn1, i;
int p_new_size;
int *polyA, *polyB, *resultPoly;
int resultCount;

p_count1 = window->getPointCount();
p_count2 = poly->getPointCount();
p_new_size = (p_count1 + p_count2) * 2;
polyA = new int[p_new_size * 3 * sizeof(int)];
polyB = new int[p_new_size * 3 * sizeof(int)];
resultPoly = new int[p_new_size * 2 * sizeof(int)];
for (psn1 = 0; psn1 < p_count1; psn1++)
{
pt = window->getPointAt(psn1);
*(polyA + (3 * psn1)) = pt->getX();
*(polyA + (3 * psn1) + 1) = pt->getY();
*(polyA + (3 * psn1) + 2) = 0;
}
for (psn1 = 0; psn1 < p_count2; psn1++)
{
pt = poly->getPointAt(psn1);
*(polyB + (3 * psn1)) = pt->getX();
*(polyB + (3 * psn1) + 1) = pt->getY();
*(polyB + (3 * psn1) + 2) = 0;
}
for (psn1 = 0; psn1 < p_new_size; psn1++)
{
*(resultPoly + (2 * psn1)) = 255;
*(resultPoly + (2 * psn1) + 1) = 255;
}

resultCount = processPolygon->processPolygons(polyA, polyB, p_count1,
                                              p_count2, resultPoly);
if (resultCount > 0)
{
p_new = new Polygon(0, resultCount);
psn1 = 0;
for (i = 0; i < resultCount; i++)
{
p_new2 = new Polygon(p_new_size, 0);
p_new->addNested(p_new2);
while  ((*(resultPoly + (2 * psn1)) != 255) &&( psn1 < p_new_size))
{
pt = new Point(*(resultPoly + (2 * psn1)),
               *(resultPoly + (2 * psn1) + 1));
p_new2->addPoint(pt);
psn1++;
}
psn1++;
}
}
else
```

```
p_new = 0;

delete polyA;
delete polyB;
delete resultPoly;

return p_new;
};

ClipPolygons::ClipPolygons()
{
processPolygon = new ProcessPolygons();
}


ClipPolygons::~ClipPolygons()
{
delete processPolygon;
}




#ifndef __PROCESSPOLYGONS_H
#define __PROCESSPOLYGONS_H

#include "ClipConstants.h"
#include "ProcessLines.h"

class ProcessPolygons
{
public:
////////////////////////////////////////////////////
// public attributes
////////////////////////////////////////////////////



////////////////////////////////////////////////////
// public operations
////////////////////////////////////////////////////

int processPolygons(int *poly1, int *poly2, int count1, int count2,
                    int *resultPoly);
ProcessPolygons();
~ProcessPolygons();

private:
////////////////////////////////////////////////////
// private attributes
////////////////////////////////////////////////////

ProcessLines *processLines;
```

```
//////////////////////////////////////////////////
// private operation
//////////////////////////////////////////////////

void findIntersections(int *poly1, int *poly2, int *count1, int *count2,
                    int *resultPoly, int *countR);
int findEnterPoint(int *poly, int count);
void swapIntegers(int *v1, int *v2);
void swapPointers(int **p1, int **p2);
void insertPoint(int x, int y, int *poly, int psn, int *pcount, int status);
void insertResultPoint(int x, int y, int *poly, int psn, int *pcount);
bool addPoint(int x, int y, int *poly, int *pcount);
void setStatus(int *poly, int psn, int status);
int getStatus(int *poly,int psn);
void insertInsidePoints(int *poly1, int *poly2, int max1, int max2,
                        int *resultPoly, int *maxR);
int resultPolygonCount(int *poly, int count);
void listPolygon(int *poly, int count);
void listPolygonR(int *poly, int count);
};
#endif



#include "ProcessPolygons.h"


int ProcessPolygons::processPolygons(int *poly1, int *poly2, int count1,
                                  int count2, int *resultPoly)
{
int max1 = count1, max2 = count2;
int resCount = 0;
int resultCount = 0;
findIntersections(poly1, poly2, &max1, &max2, resultPoly, &resCount);
insertInsidePoints(poly1, poly2, max1, max2, resultPoly, &resCount);
resultCount =  resultPolygonCount(resultPoly, resCount);

return resultCount;
}

void ProcessPolygons::findIntersections(int *poly1, int *poly2, int *count1,
                                    int *count2, int *resultPoly, int *countR)
{
int resultCount = 0;
int resCount = 0;
int result;
int lastResult = ADD_NONE;
int lastCount1 = *count1 - 1, lastCount2 = *count2 - 1;
int max1 = *count1, max2 = *count2;
int iterationCount;
int *lPoly1 = poly1, *lPoly2 = poly2;
bool swapFlag = false;
```

```
bool intersectFlag = true;
int newX =  255, newY =  255;
int i, j;
int p1X, p1Y, p2X, p2Y, p3X, p3Y, p4X, p4Y;
int startX = 255, startY = 255;


while (intersectFlag)
{
intersectFlag = false;
iterationCount = 0;
i = (lastCount1 + 1) % max1;
while (i != lastCount1 && iterationCount <= max1)
{
{
p1X = *(lPoly1 + (3 * i));
p1Y = *(lPoly1 + (3 * i) + 1);
p2X = *(lPoly1 + (3 * ((i + 1) % max1)));
p2Y = *(lPoly1 + (3 * ((i + 1) % max1)) + 1);
j = (lastCount2 + 1) % max2;
while (j != lastCount2)
{

newX = 255;
newY = 255;

{
p3X = *(lPoly2 + (3 * j));
p3Y = *(lPoly2 + (3 * j) + 1);
p4X = *(lPoly2 + (3 * ((j + 1) % max2)));
p4Y = *(lPoly2 + (3 * ((j + 1) % max2)) + 1);

result = processLines->processLinePair(p1X, p1Y, p2X, p2Y, p3X,
                                       p3Y, p4X, p4Y);

if (result == ADD_NEW_POINT_ENTER)
{
intersectFlag = true;
lastResult = ADD_NEW_POINT_ENTER;
newX = processLines->getNewX();
newY = processLines->getNewY();
addPoint(newX, newY, resultPoly, &resCount);
insertPoint(newX, newY, lPoly1, i, &max1, INTERSECT_ENTER);
insertPoint(newX, newY, lPoly2, j, &max2, INTERSECT_LEAVE);
resultCount++;
lastCount1 = i;
lastCount2 = j;
startX = newX;
startY = newY;
break;
}
else if (result == ADD_NEW_POINT_LEAVE)
{
```

```
intersectFlag = true;
lastResult = ADD_NEW_POINT_LEAVE;
newX = processLines->getNewX();
newY = processLines->getNewY();
addPoint(newX, newY, resultPoly, &resCount);
swapFlag = true;
insertPoint(newX, newY, lPoly1, i, &max1, INTERSECT_LEAVE);
insertPoint(newX, newY, lPoly2, j, &max2, INTERSECT_ENTER);
lastCount1 = i;
lastCount2 = j;
break;
}
else
{
if (p2X == p4X && p2Y == p4Y && p2X == startX && p2Y == startY )
{
resCount++;
startX = 255;
startY = 255;
}
}
}

j = (j + 1) % max2;
}
if (swapFlag)
break;
}
iterationCount = iterationCount + 1;
i = (i + 1) % max1;
}
if (swapFlag)
{
swapFlag = false;
swapPointers(&lPoly1, &lPoly2);
swapIntegers(&max1, &max2);
swapIntegers(&lastCount1, &lastCount2);
lastResult = ADD_NEW_POINT_ENTER;
}
}

if (lPoly1 == poly1)
{
*count1 = max1;
*count2 = max2;
}
else
{
*count1 = max2;
*count2 = max1;
}
*countR = resCount;
}
```

```
void ProcessPolygons::insertInsidePoints(int *poly1, int *poly2, int max1,
                                         int max2, int *resultPoly, int *maxR)
{
int enterX, enterY;
int i, k;

while ((i = findEnterPoint(poly1, max1)) != max1 + 1)
{
setStatus(poly1, i, INTERSECT_DONE);
enterX = *(poly1 + (3 * i));
enterY = *(poly1 + (3 * i) + 1);
i = (i + 1) % max1;
k = 0;
for (k = 0; *(resultPoly + (2 * k)) != enterX ||
            *(resultPoly + (2 * k) + 1) != enterY; k++);
while (*(poly1 + (3 * i) + 2) != INTERSECT_LEAVE)
{
insertResultPoint(*(poly1 + (3 * i)),  *(poly1 + (3 * i) + 1),
                  resultPoly, k, maxR);
k = k + 1;
i = (i + 1) % max1;
}
}

while ((i = findEnterPoint(poly2, max2)) != max2 + 1)
{
setStatus(poly2, i, INTERSECT_DONE);
enterX = *(poly2 + (3 * i));
enterY = *(poly2 + (3 * i) + 1);
i = (i + 1) % max2;
k = 0;
for (k = 0; *(resultPoly + (2 * k)) != enterX ||
            *(resultPoly + (2 * k) + 1) != enterY; k++);
while (*(poly2 + (3 * i) + 2) != INTERSECT_LEAVE)
{
insertResultPoint(*(poly2 + (3 * i)),  *(poly2 + (3 * i) + 1),
                  resultPoly, k, maxR);
k = k + 1;
i = (i + 1) % max2;
}
}
}

int ProcessPolygons::findEnterPoint(int *poly, int count)
{
int i;

for (i = 0; i < count; i++)
if (*(poly + (3 * i) + 2) == INTERSECT_ENTER)
return i;
return count + 1;
}
```

```
int ProcessPolygons::resultPolygonCount(int *poly, int pcount)
{
int i;
int resultCount = 0;

for (i = 0; i < pcount; i++)
if ((*(poly + (2 * i)) != 255) && (*(poly + (2 * (i + 1))) == 255))
resultCount = resultCount + 1;

return resultCount;
}

void ProcessPolygons::swapIntegers(int *v1, int *v2)
{
int t;

t = *v1;
*v1 = *v2;
*v2 = t;
}

void ProcessPolygons::swapPointers(int **p1, int **p2)
{
int *t;

t = *p1;
*p1 = *p2;
*p2 = t;
}


void ProcessPolygons::insertPoint(int x, int y,int *poly,int psn, int *pcount,
                                  int status)
{
int start = psn + 1;
int end = *pcount;
int i;

for (i = end - 1; i >= start; i--)
{
*(poly + (3 * (i + 1))) = *(poly + (3 * i));
*(poly + (3 * (i + 1)) + 1) = *(poly + (3 * i) + 1);
*(poly + (3 * (i + 1)) + 2) = *(poly + (3 * i) + 2);
}
*(poly + (3 * start)) = x;
*(poly + (3 * start) + 1) = y;
setStatus( poly, start, status);
*pcount = end + 1;
}

void ProcessPolygons::insertResultPoint(int x, int y,int *poly,int psn,
                                  int *pcount)
```

```
{
int start = psn + 1;
int end = *pcount;
int i;

for (i = end - 1; i >= start; i--)
{
*(poly + (2 * (i + 1))) = *(poly + (2 * i));
*(poly + (2 * (i + 1)) + 1) = *(poly + (2 * i) + 1);
}
*(poly + (2 * start)) = x;
*(poly + (2 * start) + 1) = y;
*pcount = end + 1;
}



bool ProcessPolygons::addPoint(int x, int y,int *poly, int *pcount)
{
int end = *pcount;
int i;
bool found = false;

for (i = 0; i < end; i++)
if ((*(poly + (2 * i)) == x) && (*(poly + (2 * i) + 1) == y))
found = true;

if (!found)
{
*(poly + (2 * i)) = x;
*(poly + (2 * i) + 1) = y;
}
*pcount = i + 1;

return found;
}

void ProcessPolygons::setStatus(int *poly, int psn, int status)
{
*(poly + (3 * psn) + 2) = status;
}

int ProcessPolygons::getStatus(int *poly, int psn)
{
return *(poly + (3 * psn) + 2);
}


ProcessPolygons::ProcessPolygons()
{
processLines = new ProcessLines();
}
```

```
ProcessPolygons::~ProcessPolygons()
{
delete processLines;
}
```

```
#ifndef __PROCESSLINES_H
#define __PROCESSLINES_H

#include "ClipConstants.h"


class ProcessLines
{
public:
/////////////////////////////////////////////////////
// public attributes
/////////////////////////////////////////////////////


/////////////////////////////////////////////////////
// public operations
/////////////////////////////////////////////////////


int processLinePair(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y,
                    int p4X, int p4Y);

int getNewX();
int getNewY();

private:
/////////////////////////////////////////////////////
// private attributes
/////////////////////////////////////////////////////

int newX1;
int newY1;

/////////////////////////////////////////////////////
// private operations
/////////////////////////////////////////////////////

int checkInside(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y);

float computeT0(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y,
                  int p4X, int p4Y);

float computeU0(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y,
```

```
                    int p4X, int p4Y, float t0);

int checkIntersect(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y,
                    int p4X, int p4Y, float t0, float u0);


};

#endif
```

```
#include <math.h>
#include "ProcessLines.h"

int ProcessLines::processLinePair(int p1X, int p1Y, int p2X, int p2Y, int p3X,
                        int p3Y, int p4X, int p4Y)
{
int result;
int inside1, inside2;
int intersect;
float t0;
float u0;

newX1 = 255;
newY1 = 255;
inside1 = checkInside(p3X, p3Y, p4X, p4Y, p1X, p1Y);
inside2 = checkInside(p3X, p3Y, p4X, p4Y, p2X, p2Y);
t0 = computeT0(p1X, p1Y, p2X, p2Y, p3X, p3Y, p4X, p4Y);
u0 = computeU0(p1X, p1Y, p2X, p2Y, p3X, p3Y, p4X, p4Y, t0);
intersect = checkIntersect(p1X, p1Y, p2X, p2Y, p3X, p3Y, p4X, p4Y, t0, u0);


if (inside1 == OUTSIDE && inside2 == INSIDE && intersect == INTERSECT)
{
newX1 = (int)round(p1X + ((p2X - p1X) * t0));
newY1 = (int)round(p1Y + ((p2Y - p1Y) * t0));
result = ADD_NEW_POINT_ENTER;
}
else if (inside1 == INSIDE && inside2 == OUTSIDE && intersect == INTERSECT)
{
newX1 = (int)(p1X + ((p2X - p1X) * t0));
newY1 = (int)(p1Y + ((p2Y - p1Y) * t0));
result = ADD_NEW_POINT_LEAVE;
}
else
result = ADD_NONE;

return result;
}
```

```
int ProcessLines::checkInside(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y)
{
int result = INSIDE;
int xn, yn;
int xt, yt;
int d;

xn = -(p2Y - p1Y);
yn = p2X - p1X;
xt = p3X - p1X;
yt = p3Y - p1Y;
d = xn * xt + yn * yt;
if (d == 0)
result = ON_LINE;
else if (d > 0)
result = OUTSIDE;
return result;
}


float ProcessLines::computeT0(int p1X, int p1Y, int p2X, int p2Y, int p3X,
                              int p3Y, int p4X, int p4Y)
{

return ((float)((p3X - p1X) * (p4Y - p3Y) - (p3Y - p1Y) * (p4X -p3X)))
       /((float)((p2X - p1X) * (p4Y - p3Y) - (p2Y - p1Y) * (p4X - p3X)));
}

float ProcessLines::computeU0(int p1X, int p1Y, int p2X, int p2Y, int p3X,
                              int p3Y, int p4X, int p4Y, float t0)
{
if ((p4X - p3X) != 0)
return ((float)((p1X + (p2X - p1X) * t0) - p3X)) / (float)(p4X - p3X);
else if ((p4Y - p3Y) != 0)
return ((float)((p1Y + (p2Y - p1Y) * t0) - p3Y)) / (float)(p4Y -  p3Y);
else
return 100.0;
}

int ProcessLines::checkIntersect(int p1X, int p1Y, int p2X, int p2Y, int p3X,
                                 int p3Y, int p4X, int p4Y, float t0, float u0)
{
int result = NO_INTERSECT;
int d;

d = (p2X - p1X) * (p4Y - p3Y) - (p2Y - p1Y) * (p4X - p3X);
if (d != 0)
{
if (t0 >= 0.0 && 1.0 >= t0)
{
if (0.0 <= u0 && u0 <= 1.0)
```

```
{
result = INTERSECT;
}
}
}

return result;
}


int ProcessLines::getNewX()
{
return newX1;
}

int ProcessLines::getNewY()
{
return newY1;
}


// C++ header file "Draw_Clipped.h" for class Draw_Clipped generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).




#ifndef Draw_Clipped_Im84629e2m10a560450ccmm7a8b_H
#define Draw_Clipped_Im84629e2m10a560450ccmm7a8b_H

#include "Polygon.h"

class DrawClipped
{
public:
////////////////////////////////////////////////////
// public operations
////////////////////////////////////////////////////

void draw(Polygon *poly);


};

#endif // Draw_Clipped_Im84629e2m10a560450ccmm7a8b_H


// C++ implementation file "Draw_Clipped.cpp" for class Draw_Clipped generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).
```

```
#include <iostream>
#include "DrawClipped.h"

void DrawClipped::draw(Polygon *poly)
{
int i, j;
int p_count, n_count;
int x, y;
Polygon *p_temp;
Point *pt;

p_count = poly->getPointCount();
n_count = poly->getNestedCount();
std::cout << "Polygon\n";
for (i = 0; i < p_count; i++)
{
pt = poly->getPointAt(i);
x = pt->getX();
y = pt->getY();
std::cout << "(" << x << "," << y << ")\n";
}

for (j = 0; j < n_count; j++)
{
p_temp = poly->getNestedAt(j);
p_count = p_temp->getPointCount();
std::cout << "    Output Polygon " << j << "\n";
for (i = 0; i < p_count; i++)
{
pt = p_temp->getPointAt(i);
x = pt->getX();
y = pt->getY();
std::cout << "        (" <<x << "," << y << ")\n";
}
}
}
```

B. Embedded Software Implementation

This Appendix contains the source code for the embedded software version of the polygon clipping system. Only the portions of the system that differ from the software-only version are included. Much of the code in the embedded software part of this implementation is copyrighted material of the Altera Corporation and is not included in this source listing for that reason. It is available from the DE2 board distribution CD [11].

```
// C++ implementation file "ClipPolygons.cpp" for class ClipPolygons generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#include <windows.h>
#include <winbase.h>
#include <iostream>
#include "ProcessPolygons.h"
#include "EasyD12.h"


int ProcessPolygons::processPolygons(int *poly1, int *poly2, int count1,
                                     int count2, int *resultPoly)
{
int max1 = count1, max2 = count2;
int resCount = 0;
int resultCount = 0;
findIntersections(poly1, poly2, &max1, &max2, resultPoly, &resCount);
insertInsidePoints(poly1, poly2, max1, max2, resultPoly, &resCount);
resultCount =  resultPolygonCount(resultPoly, resCount);

return resultCount;
}

void ProcessPolygons::findIntersections(int *poly1, int *poly2, int *count1,
                                        int *count2, int *resultPoly, int *countR)
{
unsigned long cpu_s_high;
unsigned long cpu_s_low;
unsigned long cpu_e_high;
unsigned long cpu_e_low;
int newXtemp;
int newYtemp;
int resultCount = 0;
int resCount = 0;
int result;
int lastResult = ADD_NONE;
int lastCount1 = *count1 - 1, lastCount2 = *count2 - 1;
int max1 = *count1, max2 = *count2;
int iterationCount;
int *lPoly1 = poly1, *lPoly2 = poly2;
bool swapFlag = false;
bool intersectFlag = true;
int newX =  255, newY =  255;
```

```
int i, j;
int p1X, p1Y, p2X, p2Y, p3X, p3Y, p4X, p4Y;
int startX = 255, startY = 255;
unsigned char sbuffer[8], rbuffer[8];


while (intersectFlag)
{
intersectFlag = false;
iterationCount = 0;
i = (lastCount1 + 1) % max1;
while (i != lastCount1 && iterationCount <= max1)
{
{
p1X = *(lPoly1 + (3 * i));
p1Y = *(lPoly1 + (3 * i) + 1);
p2X = *(lPoly1 + (3 * ((i + 1) % max1)));
p2Y = *(lPoly1 + (3 * ((i + 1) % max1)) + 1);
j = (lastCount2 + 1) % max2;
while (j != lastCount2)
{

newX = 255;
newY = 255;

{
p3X = *(lPoly2 + (3 * j));
p3Y = *(lPoly2 + (3 * j) + 1);
p4X = *(lPoly2 + (3 * ((j + 1) % max2)));
p4Y = *(lPoly2 + (3 * ((j + 1) % max2)) + 1);

__asm cpuid
__asm rdtsc
__asm mov cpu_s_high, edx
__asm mov cpu_s_low, eax

sbuffer[0] = (unsigned char)p1X;
sbuffer[1] = (unsigned char)p1Y;
sbuffer[2] = (unsigned char)p2X;
sbuffer[3] = (unsigned char)p2Y;
sbuffer[4] = (unsigned char)p3X;
sbuffer[5] = (unsigned char)p3Y;
sbuffer[6] = (unsigned char)p4X;
sbuffer[7] = (unsigned char)p4Y;

WritePort1(sbuffer, 8);
ReadPort2(rbuffer, 8);

result = (int)rbuffer[0];
newXtemp = (int)rbuffer[1];
newYtemp = (int)rbuffer[2];

__asm cpuid
```

```
__asm rdtsc
__asm mov cpu_e_high, edx
__asm mov cpu_e_low, eax
printf("DATA,   %d, %d, %d, %d, %d, %d, %d, %d, %d %d, %d\n",
       sbuffer[0], sbuffer[1], sbuffer[2], sbuffer[3],
   sbuffer[4], sbuffer[5], sbuffer[6], sbuffer[7],
   rbuffer[0], rbuffer[1], rbuffer[2]);
printf("TIMING, %lu, %lu, %lu, %lu\n", cpu_s_high, cpu_s_low,
       cpu_e_high, cpu_e_low);

if (result == ADD_NEW_POINT_ENTER)
{
intersectFlag = true;
lastResult = ADD_NEW_POINT_ENTER;
newX = newXtemp;
newY = newYtemp;
addPoint(newX, newY, resultPoly, &resCount);
insertPoint(newX, newY, lPoly1, i, &max1, INTERSECT_ENTER);
insertPoint(newX, newY, lPoly2, j, &max2, INTERSECT_LEAVE);
resultCount++;
lastCount1 = i;
lastCount2 = j;
startX = newX;
startY = newY;
break;
}
else if (result == ADD_NEW_POINT_LEAVE)
{
intersectFlag = true;
lastResult = ADD_NEW_POINT_LEAVE;
newX = newXtemp;
newY = newYtemp;
addPoint(newX, newY, resultPoly, &resCount);
swapFlag = true;
insertPoint(newX, newY, lPoly1, i, &max1, INTERSECT_LEAVE);
insertPoint(newX, newY, lPoly2, j, &max2, INTERSECT_ENTER);
lastCount1 = i;
lastCount2 = j;
break;
}
else
{
if (p2X == p4X && p2Y == p4Y && p2X == startX && p2Y == startY )
{
resCount++;
startX = 255;
startY = 255;
}
}
}

j = (j + 1) % max2;
}
```

```
if (swapFlag)
break;
}
iterationCount = iterationCount + 1;
i = (i + 1) % max1;
}
if (swapFlag)
{
swapFlag = false;
swapPointers(&lPoly1, &lPoly2);
swapIntegers(&max1, &max2);
swapIntegers(&lastCount1, &lastCount2);
lastResult = ADD_NEW_POINT_ENTER;
}
}

if (lPoly1 == poly1)
{
*count1 = max1;
*count2 = max2;
}
else
{
*count1 = max2;
*count2 = max1;
}
*countR = resCount;
}

void ProcessPolygons::insertInsidePoints(int *poly1, int *poly2, int max1,
                                         int max2, int *resultPoly, int *maxR)
{
int enterX, enterY;
int i, k;

while ((i = findEnterPoint(poly1, max1)) != max1 + 1)
{
setStatus(poly1, i, INTERSECT_DONE);
enterX = *(poly1 + (3 * i));
enterY = *(poly1 + (3 * i) + 1);
i = (i + 1) % max1;
k = 0;
for (k = 0; *(resultPoly + (2 * k)) != enterX ||
            *(resultPoly + (2 * k) + 1) != enterY; k++);
while (*(poly1 + (3 * i) + 2) != INTERSECT_LEAVE)
{
insertResultPoint(*(poly1 + (3 * i)),  *(poly1 + (3 * i) + 1),
                  resultPoly, k, maxR);
k = k + 1;
i = (i + 1) % max1;
}
}
```

```
while ((i = findEnterPoint(poly2, max2)) != max2 + 1)
{
setStatus(poly2, i, INTERSECT_DONE);
enterX = *(poly2 + (3 * i));
enterY = *(poly2 + (3 * i) + 1);
i = (i + 1) % max2;
k = 0;
for (k = 0; *(resultPoly + (2 * k)) != enterX ||
            *(resultPoly + (2 * k) + 1) != enterY; k++);
while (*(poly2 + (3 * i) + 2) != INTERSECT_LEAVE)
{
insertResultPoint(*(poly2 + (3 * i)),  *(poly2 + (3 * i) + 1),
                  resultPoly, k, maxR);
k = k + 1;
i = (i + 1) % max2;
}
}
}

int ProcessPolygons::findEnterPoint(int *poly, int count)
{
int i;

for (i = 0; i < count; i++)
if (*(poly + (3 * i) + 2) == INTERSECT_ENTER)
return i;
return count + 1;
}

int ProcessPolygons::resultPolygonCount(int *poly, int pcount)
{
int i;
int resultCount = 0;

for (i = 0; i < pcount; i++)
if ((*(poly + (2 * i)) != 255) && (*(poly + (2 * (i + 1))) == 255))
resultCount = resultCount + 1;

return resultCount;
}

void ProcessPolygons::swapIntegers(int *v1, int *v2)
{
int t;

t = *v1;
*v1 = *v2;
*v2 = t;
}

void ProcessPolygons::swapPointers(int **p1, int **p2)
{
int *t;
```

```
t = *p1;
*p1 = *p2;
*p2 = t;
}


void ProcessPolygons::insertPoint(int x, int y,int *poly,int psn, int *pcount,
                                  int status)
{
int start = psn + 1;
int end = *pcount;
int i;

for (i = end - 1; i >= start; i--)
{
*(poly + (3 * (i + 1))) = *(poly + (3 * i));
*(poly + (3 * (i + 1)) + 1) = *(poly + (3 * i) + 1);
*(poly + (3 * (i + 1)) + 2) = *(poly + (3 * i) + 2);
}
*(poly + (3 * start)) = x;
*(poly + (3 * start) + 1) = y;
setStatus( poly, start, status);
*pcount = end + 1;
}

void ProcessPolygons::insertResultPoint(int x, int y,int *poly,int psn,
                                  int *pcount)
{
int start = psn + 1;
int end = *pcount;
int i;

for (i = end - 1; i >= start; i--)
{
*(poly + (2 * (i + 1))) = *(poly + (2 * i));
*(poly + (2 * (i + 1)) + 1) = *(poly + (2 * i) + 1);
}
*(poly + (2 * start)) = x;
*(poly + (2 * start) + 1) = y;
*pcount = end + 1;
}



bool ProcessPolygons::addPoint(int x, int y,int *poly, int *pcount)
{
int end = *pcount;
int i;
bool found = false;

for (i = 0; i < end; i++)
if ((*(poly + (2 * i)) == x) && (*(poly + (2 * i) + 1) == y))
```

```
found = true;

if (!found)
{
*(poly + (2 * i)) = x;
*(poly + (2 * i) + 1) = y;
}
*pcount = i + 1;

return found;
}

void ProcessPolygons::setStatus(int *poly, int psn, int status)
{
*(poly + (3 * psn) + 2) = status;
}

int ProcessPolygons::getStatus(int *poly, int psn)
{
return *(poly + (3 * psn) + 2);
}


ProcessPolygons::ProcessPolygons()
{
// processLines = new ProcessLines();
}


ProcessPolygons::~ProcessPolygons()
{
// delete processLines;
}




#ifndef __PROCESSLINES_H
#define __PROCESSLINES_H

#include "ClipConstants.h"



int processLinePair(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y,
                    int p4X, int p4Y);

int getNewX();
int getNewY();

int checkInside(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y);

float computeT0(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y,
```

```
                    int p4X, int p4Y);

float computeU0(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y,
                    int p4X, int p4Y, float t0);

int checkIntersect(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y,
                    int p4X, int p4Y, float t0, float u0);

#endif
```

```
#include <math.h>
#include "ProcessLines.h"

static int newX1;
static int newY1;

int processLinePair(int p1X, int p1Y, int p2X, int p2Y, int p3X,
                     int p3Y, int p4X, int p4Y)
{
int result;
int inside1, inside2;
int intersect;
float t0;
float u0;

newX1 = 255;
newY1 = 255;
inside1 = checkInside(p3X, p3Y, p4X, p4Y, p1X, p1Y);
inside2 = checkInside(p3X, p3Y, p4X, p4Y, p2X, p2Y);
t0 = computeT0(p1X, p1Y, p2X, p2Y, p3X, p3Y, p4X, p4Y);
u0 = computeU0(p1X, p1Y, p2X, p2Y, p3X, p3Y, p4X, p4Y, t0);
intersect = checkIntersect(p1X, p1Y, p2X, p2Y, p3X, p3Y, p4X, p4Y, t0, u0);


if (inside1 == OUTSIDE && inside2 == INSIDE && intersect == INTERSECT)
{
newX1 = (int)round(p1X + ((p2X - p1X) * t0));
newY1 = (int)round(p1Y + ((p2Y - p1Y) * t0));
result = ADD_NEW_POINT_ENTER;
}
else if (inside1 == INSIDE && inside2 == OUTSIDE && intersect == INTERSECT)
{
newX1 = (int)(p1X + ((p2X - p1X) * t0));
newY1 = (int)(p1Y + ((p2Y - p1Y) * t0));
result = ADD_NEW_POINT_LEAVE;
}
else
result = ADD_NONE;

return result;
```

```
}



int checkInside(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y)
{
int result = INSIDE;
int xn, yn;
int xt, yt;
int d;

xn = -(p2Y - p1Y);
yn = p2X - p1X;
xt = p3X - p1X;
yt = p3Y - p1Y;
d = xn * xt + yn * yt;
if (d == 0)
result = ON_LINE;
else if (d > 0)
result = OUTSIDE;
return result;
}


float computeT0(int p1X, int p1Y, int p2X, int p2Y, int p3X,
                int p3Y, int p4X, int p4Y)
{

return ((float)((p3X - p1X) * (p4Y - p3Y) - (p3Y - p1Y) * (p4X -p3X)))
       /((float)((p2X - p1X) * (p4Y - p3Y) - (p2Y - p1Y) * (p4X - p3X)));
}

float computeU0(int p1X, int p1Y, int p2X, int p2Y, int p3X,
                int p3Y, int p4X, int p4Y, float t0)
{
if ((p4X - p3X) != 0)
return ((float)((p1X + (p2X - p1X) * t0) - p3X)) / (float)(p4X - p3X);
else if ((p4Y - p3Y) != 0)
return ((float)((p1Y + (p2Y - p1Y) * t0) - p3Y)) / (float)(p4Y -  p3Y);
else
return 100.0;
}

int checkIntersect(int p1X, int p1Y, int p2X, int p2Y, int p3X,
                   int p3Y, int p4X, int p4Y, float t0, float u0)
{
int result = NO_INTERSECT;
int d;

d = (p2X - p1X) * (p4Y - p3Y) - (p2Y - p1Y) * (p4X - p3X);
if (d != 0)
{
if (t0 >= 0.0 && 1.0 >= t0)
```

```
{
if (0.0 <= u0 && u0 <= 1.0)
{
result = INTERSECT;
}
}
}

return result;
}


int getNewX()
{
return newX1;
}

int getNewY()
{
return newY1;
}
```

C. SystemC Hardware Simulation Implementation

This Appendix contains the source code for the SystemC simulation of the hardware/software co-designed version of the polygon clipping application.

```
#ifndef __CLIPCONSTANTS_H
#define __CLIPCONSTANTS_H

#define NO_INTERSECT (sc_uint<2>)0
#define INTERSECT    (sc_uint<2>)1

#define INSIDE  (sc_uint<2>)1
#define OUTSIDE (sc_uint<2>)2
#define ON_LINE (sc_uint<2>)3

#define ADD_NONE            (sc_uint<2>)0
#define ADD_NEW_POINT_ENTER (sc_uint<2>)1
#define ADD_NEW_POINT_LEAVE (sc_uint<2>)2

#define NOT_USED        (sc_uint<2>)0
#define INTERSECT_ENTER (sc_uint<2>)1
#define INTERSECT_LEAVE (sc_uint<2>)2
#define INTERSECT_DONE  (sc_uint<2>)3

#define BUFF_SIZE (sc_uint<8>)8

#endif
```

```
#include "CheckInside.h"
#include "CheckIntersect.h"
#include "ComputeT0.h"
#include "ComputeU0.h"
#include "ComputeCoordinate.h"
#include "ProcessLines.h"
#include "Test_Drvr.h"
#include "Clipper.h"

char *fName;
Test_Drvr *tDrvr;

int sc_main(int argc, char *argv[])
{
scv_random::set_global_seed(101);

sc_clock clock("CLOCK", 10, 0.5, 0.0);

// CheckInside to ProcessLines
sc_signal<sc_bv<9> > ins_p1Xs;
sc_signal<sc_bv<9> > ins_p1Ys;
```

```
sc_signal<sc_bv<9> > ins_p2Xs;
sc_signal<sc_bv<9> > ins_p2Ys;
sc_signal<sc_bv<9> > ins_p3Xs;
sc_signal<sc_bv<9> > ins_p3Ys;
sc_signal<sc_logic > ins_data_sets;
sc_signal<sc_bv<2> > ins_result_outs;
sc_signal<sc_logic > ins_result_sets;

// CheckIntersect to ProcessLines
sc_signal<sc_bv<9> > int_p1Xs;
sc_signal<sc_bv<9> > int_p1Ys;
sc_signal<sc_bv<9> > int_p2Xs;
sc_signal<sc_bv<9> > int_p2Ys;
sc_signal<sc_bv<9> > int_p3Xs;
sc_signal<sc_bv<9> > int_p3Ys;
sc_signal<sc_bv<9> > int_p4Xs;
sc_signal<sc_bv<9> > int_p4Ys;
sc_signal<float >    int_t0s;
sc_signal<float >    int_u0s;
sc_signal<sc_logic > int_data_sets;
sc_signal<sc_bv<2> > int_result_outs;
sc_signal<sc_logic > int_result_sets;

// ComputeT0 to ProcessLines
sc_signal<sc_bv<9> > t0_p1Xs;
sc_signal<sc_bv<9> > t0_p1Ys;
sc_signal<sc_bv<9> > t0_p2Xs;
sc_signal<sc_bv<9> > t0_p2Ys;
sc_signal<sc_bv<9> > t0_p3Xs;
sc_signal<sc_bv<9> > t0_p3Ys;
sc_signal<sc_bv<9> > t0_p4Xs;
sc_signal<sc_bv<9> > t0_p4Ys;
sc_signal<sc_logic > t0_data_sets;
sc_signal<float >    t0_result_outs;
sc_signal<sc_logic > t0_result_sets;

// ComputeU0 to ProcessLines
sc_signal<sc_bv<9> > u0_p1Xs;
sc_signal<sc_bv<9> > u0_p1Ys;
sc_signal<sc_bv<9> > u0_p2Xs;
sc_signal<sc_bv<9> > u0_p2Ys;
sc_signal<sc_bv<9> > u0_p3Xs;
sc_signal<sc_bv<9> > u0_p3Ys;
sc_signal<sc_bv<9> > u0_p4Xs;
sc_signal<sc_bv<9> > u0_p4Ys;
sc_signal<float >    u0_t0s;
sc_signal<sc_logic > u0_data_sets;
sc_signal<float >    u0_result_outs;
sc_signal<sc_logic > u0_result_sets;

// ComputeCoordinate to ProcessLines
sc_signal<sc_bv<9> > cc_v1s;
sc_signal<sc_bv<9> > cc_v2s;
```

```
sc_signal<float >    cc_t0s;
sc_signal<sc_logic > cc_data_sets;
sc_signal<sc_bv<9> > cc_result_outs;
sc_signal<sc_logic > cc_result_sets;

// ProcessLines to Test_Drvr;
sc_signal<sc_bv<9> > pl_p1Xs;
sc_signal<sc_bv<9> > pl_p1Ys;
sc_signal<sc_bv<9> > pl_p2Xs;
sc_signal<sc_bv<9> > pl_p2Ys;
sc_signal<sc_bv<9> > pl_p3Xs;
sc_signal<sc_bv<9> > pl_p3Ys;
sc_signal<sc_bv<9> > pl_p4Xs;
sc_signal<sc_bv<9> > pl_p4Ys;
sc_signal<sc_logic > pl_data_sets;
sc_signal<sc_bv<2> > pl_result_outs;
sc_signal<sc_bv<9> > pl_resultX_outs;
sc_signal<sc_bv<9> > pl_resultY_outs;
sc_signal<sc_logic > pl_result_sets;

// Hook up CheckInside
CheckInside checkInside("Inside");
checkInside.clk(clock);
checkInside.p1Xs(ins_p1Xs);
checkInside.p1Ys(ins_p1Ys);
checkInside.p2Xs(ins_p2Xs);
checkInside.p2Ys(ins_p2Ys);
checkInside.p3Xs(ins_p3Xs);
checkInside.p3Ys(ins_p3Ys);
checkInside.data_sets(ins_data_sets);
checkInside.result_outs(ins_result_outs);
checkInside.result_sets(ins_result_sets);

// Hook up CheckIntersect
CheckIntersect checkIntersect("Intersect");
checkIntersect.clk(clock);
checkIntersect.p1Xs(int_p1Xs);
checkIntersect.p1Ys(int_p1Ys);
checkIntersect.p2Xs(int_p2Xs);
checkIntersect.p2Ys(int_p2Ys);
checkIntersect.p3Xs(int_p3Xs);
checkIntersect.p3Ys(int_p3Ys);
checkIntersect.p4Xs(int_p4Xs);
checkIntersect.p4Ys(int_p4Ys);
checkIntersect.t0s(int_t0s);
checkIntersect.u0s(int_u0s);
checkIntersect.data_sets(int_data_sets);
checkIntersect.result_outs(int_result_outs);
checkIntersect.result_sets(int_result_sets);

// Hook ComputeT0
ComputeT0 computeT0("T0");
computeT0.clk(clock);
```

```
computeT0.p1Xs(t0_p1Xs);
computeT0.p1Ys(t0_p1Ys);
computeT0.p2Xs(t0_p2Xs);
computeT0.p2Ys(t0_p2Ys);
computeT0.p3Xs(t0_p3Xs);
computeT0.p3Ys(t0_p3Ys);
computeT0.p4Xs(t0_p4Xs);
computeT0.p4Ys(t0_p4Ys);
computeT0.data_sets(t0_data_sets);
computeT0.result_outs(t0_result_outs);
computeT0.result_sets(t0_result_sets);

// Hook ComputeU0
ComputeU0 computeU0("U0");
computeU0.clk(clock);
computeU0.p1Xs(u0_p1Xs);
computeU0.p1Ys(u0_p1Ys);
computeU0.p2Xs(u0_p2Xs);
computeU0.p2Ys(u0_p2Ys);
computeU0.p3Xs(u0_p3Xs);
computeU0.p3Ys(u0_p3Ys);
computeU0.p4Xs(u0_p4Xs);
computeU0.p4Ys(u0_p4Ys);
computeU0.t0s(u0_t0s);
computeU0.data_sets(u0_data_sets);
computeU0.result_outs(u0_result_outs);
computeU0.result_sets(u0_result_sets);

// Hook up ComputeCoordinate
ComputeCoordinate computeCoordinate("Coordinate");
computeCoordinate.clk(clock);
computeCoordinate.v1s(cc_v1s);
computeCoordinate.v2s(cc_v2s);
computeCoordinate.t0s(cc_t0s);
computeCoordinate.data_sets(cc_data_sets);
computeCoordinate.result_outs(cc_result_outs);
computeCoordinate.result_sets(cc_result_sets);

// Hook up Test_Drvr
Test_Drvr testDrvr("Driver");
testDrvr.clk(clock);
testDrvr.p1Xs(pl_p1Xs);
testDrvr.p1Ys(pl_p1Ys);
testDrvr.p2Xs(pl_p2Xs);
testDrvr.p2Ys(pl_p2Ys);
testDrvr.p3Xs(pl_p3Xs);
testDrvr.p3Ys(pl_p3Ys);
testDrvr.p4Xs(pl_p4Xs);
testDrvr.p4Ys(pl_p4Ys);
testDrvr.data_sets(pl_data_sets);
testDrvr.result_outs(pl_result_outs);
testDrvr.resultX_outs(pl_resultX_outs);
testDrvr.resultY_outs(pl_resultY_outs);
```

```
testDrvr.result_sets(pl_result_sets);

// Hook up ProcessLines
ProcessLines processLines("Process");
processLines.clk(clock);
processLines.ins_p1Xs(ins_p1Xs);
processLines.ins_p1Ys(ins_p1Ys);
processLines.ins_p2Xs(ins_p2Xs);
processLines.ins_p2Ys(ins_p2Ys);
processLines.ins_p3Xs(ins_p3Xs);
processLines.ins_p3Ys(ins_p3Ys);
processLines.ins_data_sets(ins_data_sets);
processLines.ins_result_outs(ins_result_outs);
processLines.ins_result_sets(ins_result_sets);
processLines.int_p1Xs(int_p1Xs);
processLines.int_p1Ys(int_p1Ys);
processLines.int_p2Xs(int_p2Xs);
processLines.int_p2Ys(int_p2Ys);
processLines.int_p3Xs(int_p3Xs);
processLines.int_p3Ys(int_p3Ys);
processLines.int_p4Xs(int_p4Xs);
processLines.int_p4Ys(int_p4Ys);
processLines.int_t0s(int_t0s);
processLines.int_u0s(int_u0s);
processLines.int_data_sets(int_data_sets);
processLines.int_result_outs(int_result_outs);
processLines.int_result_sets(int_result_sets);
processLines.t0_p1Xs(t0_p1Xs);
processLines.t0_p1Ys(t0_p1Ys);
processLines.t0_p2Xs(t0_p2Xs);
processLines.t0_p2Ys(t0_p2Ys);
processLines.t0_p3Xs(t0_p3Xs);
processLines.t0_p3Ys(t0_p3Ys);
processLines.t0_p4Xs(t0_p4Xs);
processLines.t0_p4Ys(t0_p4Ys);
processLines.t0_data_sets(t0_data_sets);
processLines.t0_result_outs(t0_result_outs);
processLines.t0_result_sets(t0_result_sets);
processLines.u0_p1Xs(u0_p1Xs);
processLines.u0_p1Ys(u0_p1Ys);
processLines.u0_p2Xs(u0_p2Xs);
processLines.u0_p2Ys(u0_p2Ys);
processLines.u0_p3Xs(u0_p3Xs);
processLines.u0_p3Ys(u0_p3Ys);
processLines.u0_p4Xs(u0_p4Xs);
processLines.u0_p4Ys(u0_p4Ys);
processLines.u0_t0s(u0_t0s);
processLines.u0_data_sets(u0_data_sets);
processLines.u0_result_outs(u0_result_outs);
processLines.u0_result_sets(u0_result_sets);
processLines.cc_v1s(cc_v1s);
processLines.cc_v2s(cc_v2s);
processLines.cc_t0s(cc_t0s);
```

```
processLines.cc_data_sets(cc_data_sets);
processLines.cc_result_outs(cc_result_outs);
processLines.cc_result_sets(cc_result_sets);
processLines.p1Xs(pl_p1Xs);
processLines.p1Ys(pl_p1Ys);
processLines.p2Xs(pl_p2Xs);
processLines.p2Ys(pl_p2Ys);
processLines.p3Xs(pl_p3Xs);
processLines.p3Ys(pl_p3Ys);
processLines.p4Xs(pl_p4Xs);
processLines.p4Ys(pl_p4Ys);
processLines.data_sets(pl_data_sets);
processLines.result_outs(pl_result_outs);
processLines.resultX_outs(pl_resultX_outs);
processLines.resultY_outs(pl_resultY_outs);
processLines.result_sets(pl_result_sets);

if (argc != 2)
{
cout << "Usage run.x <data_file_name>\n";
return 1;
}
fName = argv[1];
tDrvr = &testDrvr;

Clipper clipper("Clipper");
clipper.clk(clock);

sc_start(10000000);

return 0;
}
```

```
#ifndef __POINT_H
#define __POINT_H

class Point
{
public:

int x;
int y;

Point(int pX, int pY);

int getX();

int getY();
```

```
};

#endif
```

```
// C++ implementation file "Point.cpp" for class Point generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#include "Point.h"

Point::Point(int pX, int pY)
{
x = pX;
y = pY;
}


int Point::getX()
{
return x;
}


int Point::getY()
{
return y;
}
```

```
// C++ header file "Shape.h" for class Shape generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#ifndef Shape_Im84629e2m10a560450ccmm7dad_H
#define Shape_Im84629e2m10a560450ccmm7dad_H

#include "Point.h"

class Shape
{
public:
////////////////////////////////////////////////////
// public attributes
////////////////////////////////////////////////////
```

```
Point **points;

int pointPosition;

int pointCount;

int numberPoints;


//////////////////////////////////////////////////
// public operations
//////////////////////////////////////////////////

void addPoint(Point *p);

Point *getPoint();

Point *getPointPrior();

int getPointCount();

int getPointPosition();

Point *getPointAt(int psn);

Point *getPointPriorAt(int psn);

Point *getPointNextAt(int psn);

void pointPositionReset();

void initShape(int numPoints);

Shape();

~Shape();

};

#endif // Shape_Im84629e2m10a560450ccmm7dad_H
```

```
// C++ implementation file "Shape.cpp" for class Shape generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#include "Shape.h"
```

```
void Shape::addPoint(Point *p)
{
if (pointCount < numberPoints)
{
points[pointCount] = p;
pointCount++;
}
}


Point *Shape::getPoint()
{
Point *temp;
temp = points[pointPosition];
pointPosition = (pointPosition + 1) % pointCount;
return temp;
}


Point *Shape::getPointPrior()
{
Point *temp;
int psn;

psn = pointPosition - 2;
if (psn < 0)
psn = pointCount + psn;
temp = points[psn];
return temp;
}


int Shape::getPointCount()
{
return pointCount;
}


int Shape::getPointPosition()
{
return pointPosition;
}



Point *Shape::getPointAt(int psn)
{
int p_psn = psn;
if (psn >= pointCount)
p_psn = pointCount - psn;
return points[p_psn];
}
```

```
Point *Shape::getPointPriorAt(int psn)
{
int p_psn;
p_psn = psn - 1;
if (p_psn < 0)
p_psn = pointCount - 1;
return points[p_psn];
}


Point *Shape::getPointNextAt(int psn)
{
int p_psn;
p_psn = psn + 1;
if (p_psn >= pointCount)
p_psn = p_psn - pointCount;
return points[p_psn];
}


void Shape::pointPositionReset()
{
pointPosition = 0;
}


void Shape::initShape(int numPoints)
{
numberPoints = numPoints;
pointCount = 0;
pointPosition = 0;
points = new Point *[numberPoints];
}

Shape::Shape()
{
}

Shape::~Shape()
{
int i;

for (i = 0; i < pointCount; i++)
delete points[i];
}
```

```
#ifndef __POLYGON_H
```

```
#define __POLYGON_H

#include "Shape.h"

class Polygon : public Shape
{
public:

Polygon **nestedPolygons;
int nestedCount;
int nestedPosition;
int numberNested;

void addNested(Polygon *nested);
Polygon *getNested();
Polygon *getNestedAt(int psn);
int getNestedCount();
int getNestedPosition();
void nestedPositionReset();
Polygon(int numPoints, int numNested);
~Polygon();
};

#endif
```

```
#include "Polygon.h"

void Polygon::addNested(Polygon *nested)
{
if (nestedCount < numberNested)
{
nestedPolygons[nestedCount] = nested;
nestedCount++;
}
}

Polygon *Polygon::getNested()
{
Polygon *temp;
temp = nestedPolygons[nestedPosition];
nestedPosition = (nestedPosition + 1) % nestedCount;
return temp;
}

Polygon *Polygon::getNestedAt(int psn)
{
return nestedPolygons[psn];
}
```

```
int Polygon::getNestedCount()
{
return nestedCount;
}

int Polygon::getNestedPosition()
{
return nestedPosition;
}

void Polygon::nestedPositionReset()
{
nestedPosition = 0;
}

Polygon::Polygon(int numPoints, int numNested)
{
initShape(numPoints);
numberNested = numNested;
if (numberNested > 0)
nestedPolygons = new Polygon *[numberNested];
else
nestedPolygons = 0;
nestedCount = 0;
nestedPosition = 0;
}

Polygon::~Polygon()
{
int i;

for (i = 0; i < nestedCount; i++)
delete nestedPolygons[i];
}
```

```
#ifndef Clipper_I12aa452cm10a55c27eb3mm7e73_H
#define Clipper_I12aa452cm10a55c27eb3mm7e73_H

#include "scv.h"
#include "Polygon.h"
#include "ClipPolygons.h"
#include "DrawClipped.h"
#include "Test_Drvr.h"

SC_MODULE(Clipper)
{
public:

sc_in_clk clk;
```

```
Polygon **polygons;
int polygonCount;
int polygonPosition;
int numberPolygons;

Polygon *window;
Polygon *clippedPolygon;
ClipPolygons *clipPolygons;
DrawClipped *drawClipped;

void runClipper();

void init();
void cleanup();
int loadPolygons(char *fileName);
int readAllPolygons(FILE *in_file, int numPolys);
int readOnePolygon(FILE *in_file, int numPoints, int numNested, Polygon *poly);
int readPoints(FILE *in_file, int numberPoints, Polygon *poly);

SC_CTOR(Clipper)
{
SC_CTHREAD(runClipper, clk.pos());
}

};

#endif // Clipper_I12aa452cm10a55c27eb3mm7e73_H
```

```
#include <iostream>
#include <string.h>
#include "Clipper.h"

extern char *fName;

void Clipper::runClipper()
{
int result;

init();
result = loadPolygons(fName);
if (result)
{
clippedPolygon = clipPolygons->clip(polygons[0], window);
drawClipped->draw(window);
drawClipped->draw(polygons[0]);
drawClipped->draw(clippedPolygon);
}
```

```
cleanup();
}

int Clipper::loadPolygons(char *fName)
{
FILE *in_file;
char in_line[51];
char *tok;
int pts, nsts;
char winFileName[80], polyFileName[80];

winFileName[0] = '\0';
polyFileName[0] = '\0';
int result = 1;

strcpy(winFileName, fName);
strcpy(polyFileName, fName);
strcat(winFileName, ".win");
strcat(polyFileName, ".poly");

if ((in_file = fopen(winFileName, "r")) == NULL)
{
std::cout << "File " << winFileName << " not found\n";
return 0;
}

if (fgets(in_line, 50, in_file) != NULL)
{
tok = strtok(in_line, " ,\n");
pts = atoi(tok);
tok = strtok(NULL, " ,\n");
nsts = atoi(tok);
window = new Polygon(pts, nsts);
result = readOnePolygon(in_file, pts, nsts, window);
}

fclose(in_file);

if ((in_file = fopen(polyFileName, "r")) == NULL)
{
std::cout << "File %s " <<polyFileName << " not found\n";
return 0;
}

if (result && fgets(in_line, 50, in_file) != NULL)
{
tok = strtok(in_line, " ,\n");
pts = atoi(tok);
polygonCount = pts;
polygons = new Polygon *[pts];
result = readAllPolygons(in_file, pts);
}
else
```

```
result = 0;

fclose(in_file);

return result;;
}

int Clipper::readOnePolygon(FILE *in_file, int numPoints, int numNested, Polygon *poly)
{
int result = 1;
int i;
char in_line[51];
char * tok;
int pts;
Polygon *p_temp;

result = readPoints(in_file, numPoints, poly);

for (i = 0; result && i < numNested; i++)
{
if (fgets(in_line, 50, in_file) != NULL)
{
tok = strtok(in_line, " ,\n");
pts = atoi(tok);
p_temp = new Polygon(pts, 0);
poly->addNested(p_temp);
result = readPoints(in_file, pts, p_temp);
}
else
result = 0;
}
return result;
}

int Clipper::readAllPolygons(FILE *in_file, int numPolys)
{
int i;
int p_count, n_count;
int result = 1;
char *tok;
char in_line[51];
Polygon *p_temp;

for (i = 0; result && i < numPolys; i++)
{
if (fgets(in_line, 50, in_file) != NULL)
{
tok = strtok(in_line, " ,\n");
p_count = atoi(tok);
tok = strtok(NULL, " ,\n");
n_count = atoi(tok);
p_temp = new Polygon(p_count, n_count);
polygons[i] = p_temp;
```

```
result = readOnePolygon(in_file, p_count, n_count,p_temp);
}
else
result = 0;
}
return result;
}

int Clipper::readPoints(FILE *in_file, int numberPoints, Polygon *poly)
{
int i;
Point *p_temp;
int x, y;
char *tok;
char in_line[51];
int result = 1;


for (i = 0; i < numberPoints; i++)
{
if (fgets(in_line, 50, in_file) != NULL)
{
tok = strtok(in_line, " ,\n");
x = atoi(tok);
tok = strtok(NULL, " ,\n");
y = atoi(tok);
p_temp = new Point(x, y);
poly->addPoint(p_temp);
}
else
result = 0;
}
return result;
}

void Clipper::init()
{
clipPolygons = new ClipPolygons();
drawClipped = new DrawClipped();
}

void Clipper::cleanup()
{
int i;

delete window;
delete clippedPolygon;
delete drawClipped;
delete clipPolygons;

for (i = 0; i < polygonCount; i++)
delete polygons[i];
}
```

```
// C++ header file "Draw_Clipped.h" for class Draw_Clipped generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#ifndef Draw_Clipped_Im84629e2m10a560450ccmm7a8b_H
#define Draw_Clipped_Im84629e2m10a560450ccmm7a8b_H

#include "Polygon.h"

class DrawClipped
{
public:
////////////////////////////////////////////////
// public operations
////////////////////////////////////////////////

void draw(Polygon *poly);


};

#endif // Draw_Clipped_Im84629e2m10a560450ccmm7a8b_H
```

```
// C++ implementation file "Draw_Clipped.cpp" for class Draw_Clipped generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#include <iostream>
#include "DrawClipped.h"

void DrawClipped::draw(Polygon *poly)
{
int i, j;
int p_count, n_count;
int x, y;
Polygon *p_temp;
Point *pt;

p_count = poly->getPointCount();
n_count = poly->getNestedCount();
```

```
std::cout << "Polygon\n";
for (i = 0; i < p_count; i++)
{
pt = poly->getPointAt(i);
x = pt->getX();
y = pt->getY();
std::cout << "(" << x << "," << y << ")\n";
}

for (j = 0; j < n_count; j++)
{
p_temp = poly->getNestedAt(j);
p_count = p_temp->getPointCount();
std::cout << "    Output Polygon " << j << "\n";
for (i = 0; i < p_count; i++)
{
pt = p_temp->getPointAt(i);
x = pt->getX();
y = pt->getY();
std::cout << "        (" <<x << "," << y << ")\n";
}
}
}
```

```
// C++ header file "ClipPolygons.h" for class ClipPolygons generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#ifndef ClipPolygons_I12aa452cm10a55c27eb3mm7dc0_H
#define ClipPolygons_I12aa452cm10a55c27eb3mm7dc0_H

#include "ClipConstants.h"
#include "Point.h"
#include "Polygon.h"
#include "ProcessPolygons.h"

class ClipPolygons
{
public:
////////////////////////////////////////////////////
// public attributes
////////////////////////////////////////////////////



////////////////////////////////////////////////////
// public operations
```

```
//////////////////////////////////////////////////

Polygon *clip(Polygon *window, Polygon *poly);
ClipPolygons();
~ClipPolygons();

private:
//////////////////////////////////////////////////
// private attributes
//////////////////////////////////////////////////

ProcessPolygons *processPolygon;


};
#endif
```

```
// C++ implementation file "ClipPolygons.cpp" for class ClipPolygons generated by Poseidon for UML.
// Poseidon for UML is developed by Gentleware (http://www.gentleware.com).
// Generated with velocity template engine (http://jakarta.apache.org/velocity).

#include "ClipPolygons.h"

Polygon *ClipPolygons::clip(Polygon *window, Polygon *poly)
{
Polygon *p_new, *p_new2;
Point *pt;
int p_count1, p_count2;
int psn1, i;
int p_new_size;
int *polyA, *polyB, *resultPoly;
int resultCount;

p_count1 = window->getPointCount();
p_count2 = poly->getPointCount();
p_new_size = (p_count1 + p_count2) * 2;
polyA = new int[p_new_size * 3 * sizeof(int)];
polyB = new int[p_new_size * 3 * sizeof(int)];
resultPoly = new int[p_new_size * 2 * sizeof(int)];
for (psn1 = 0; psn1 < p_count1; psn1++)
{
pt = window->getPointAt(psn1);
*(polyA + (3 * psn1)) = pt->getX();
*(polyA + (3 * psn1) + 1) = pt->getY();
*(polyA + (3 * psn1) + 2) = 0;
}
for (psn1 = 0; psn1 < p_count2; psn1++)
```

```
{
pt = poly->getPointAt(psn1);
*(polyB + (3 * psn1)) = pt->getX();
*(polyB + (3 * psn1) + 1) = pt->getY();
*(polyB + (3 * psn1) + 2) = 0;
}
for (psn1 = 0; psn1 < p_new_size; psn1++)
{
*(resultPoly + (2 * psn1)) = 255;
*(resultPoly + (2 * psn1) + 1) = 255;
}

resultCount = processPolygon->processPolygons(polyA, polyB, p_count1,
                                              p_count2, resultPoly);
if (resultCount > 0)
{
p_new = new Polygon(0, resultCount);
psn1 = 0;
for (i = 0; i < resultCount; i++)
{
p_new2 = new Polygon(p_new_size, 0);
p_new->addNested(p_new2);
while  ((*(resultPoly + (2 * psn1)) != 255) &&( psn1 < p_new_size))
{
pt = new Point(*(resultPoly + (2 * psn1)),
               *(resultPoly + (2 * psn1) + 1));
p_new2->addPoint(pt);
psn1++;
}
psn1++;
}
}
else
p_new = 0;

delete polyA;
delete polyB;
delete resultPoly;

return p_new;
};

ClipPolygons::ClipPolygons()
{
processPolygon = new ProcessPolygons();
}


ClipPolygons::~ClipPolygons()
{
delete processPolygon;
}
```

```
#ifndef __PROCESS_LINES_H
#define __PROCESS_LINES_H

#include "scv.h"

#define STATE_WAIT_SET   (sc_uint<4>)1
#define STATE_INSIDE_1   (sc_uint<4>)2
#define STATE_INSIDE_2   (sc_uint<4>)3
#define STATE_T0         (sc_uint<4>)4
#define STATE_U0         (sc_uint<4>)5
#define STATE_INTERSECT  (sc_uint<4>)6
#define STATE_NEW_X      (sc_uint<4>)7
#define STATE_NEW_Y      (sc_uint<4>)8
#define STATE_SEND_DATA  (sc_uint<4>)9
#define STATE_WAIT_UNSET (sc_uint<4>)10


SC_MODULE(ProcessLines)
{
public:
// CheckInside interface

sc_out<sc_bv<9> > ins_p1Xs;
sc_out<sc_bv<9> > ins_p1Ys;
sc_out<sc_bv<9> > ins_p2Xs;
sc_out<sc_bv<9> > ins_p2Ys;
sc_out<sc_bv<9> > ins_p3Xs;
sc_out<sc_bv<9> > ins_p3Ys;
sc_out<sc_logic > ins_data_sets;
sc_in<sc_bv<2> >  ins_result_outs;
sc_in<sc_logic >  ins_result_sets;

// CheckIntersect interface
sc_out<sc_bv<9> > int_p1Xs;
sc_out<sc_bv<9> > int_p1Ys;
sc_out<sc_bv<9> > int_p2Xs;
sc_out<sc_bv<9> > int_p2Ys;
sc_out<sc_bv<9> > int_p3Xs;
sc_out<sc_bv<9> > int_p3Ys;
sc_out<sc_bv<9> > int_p4Xs;
sc_out<sc_bv<9> > int_p4Ys;
sc_out<float >    int_t0s;
sc_out<float >    int_u0s;
sc_out<sc_logic > int_data_sets;
sc_in<sc_bv<2> >  int_result_outs;
sc_in<sc_logic >  int_result_sets;

// ComputeT0 interface
sc_out<sc_bv<9> > t0_p1Xs;
```

```
sc_out<sc_bv<9> > t0_p1Ys;
sc_out<sc_bv<9> > t0_p2Xs;
sc_out<sc_bv<9> > t0_p2Ys;
sc_out<sc_bv<9> > t0_p3Xs;
sc_out<sc_bv<9> > t0_p3Ys;
sc_out<sc_bv<9> > t0_p4Xs;
sc_out<sc_bv<9> > t0_p4Ys;
sc_out<sc_logic > t0_data_sets;
sc_in<float >     t0_result_outs;
sc_in<sc_logic >  t0_result_sets;

// ComputeU0 interface
sc_out<sc_bv<9> > u0_p1Xs;
sc_out<sc_bv<9> > u0_p1Ys;
sc_out<sc_bv<9> > u0_p2Xs;
sc_out<sc_bv<9> > u0_p2Ys;
sc_out<sc_bv<9> > u0_p3Xs;
sc_out<sc_bv<9> > u0_p3Ys;
sc_out<sc_bv<9> > u0_p4Xs;
sc_out<sc_bv<9> > u0_p4Ys;
sc_out<float >    u0_t0s;
sc_out<sc_logic > u0_data_sets;
sc_in<float >     u0_result_outs;
sc_in<sc_logic >  u0_result_sets;

// ComputeCoordinate interface
sc_out<sc_bv<9> > cc_v1s;
sc_out<sc_bv<9> > cc_v2s;
sc_out<float >    cc_t0s;
sc_out<sc_logic > cc_data_sets;
sc_in<sc_bv<9> >  cc_result_outs;
sc_in<sc_logic >  cc_result_sets;

// ProcessLines interface
sc_in_clk         clk;
sc_in<sc_bv<9> >  p1Xs;
sc_in<sc_bv<9> >  p1Ys;
sc_in<sc_bv<9> >  p2Xs;
sc_in<sc_bv<9> >  p2Ys;
sc_in<sc_bv<9> >  p3Xs;
sc_in<sc_bv<9> >  p3Ys;
sc_in<sc_bv<9> >  p4Xs;
sc_in<sc_bv<9> >  p4Ys;
sc_in<sc_logic >  data_sets;
sc_out<sc_bv<2> > result_outs;
sc_out<sc_bv<9> > resultX_outs;
sc_out<sc_bv<9> > resultY_outs;
sc_out<sc_logic > result_sets;

void processLinePair();

SC_CTOR(ProcessLines)
{
```

```
SC_CTHREAD(processLinePair, clk.pos());
}
};

#endif
```

```
#include "ProcessLines.h"
#include "ClipConstants.h"

// #define STATE_WAIT_SET   (sc_uint<4>)1
// #define STATE_INSIDE_1   (sc_uint<4>)2
// #define STATE_INSIDE_2   (sc_uint<4>)3
// #define STATE_T0         (sc_uint<4>)4
// #define STATE_U0         (sc_uint<4>)5
// #define STATE_INTERSECT  (sc_uint<4>)6
// #define STATE_NEW_X      (sc_uint<4>)7
// #define STATE_NEW_Y      (sc_uint<4>)8
// #define STATE_SEND_DATA  (sc_uint<4>)9
// #define STATE_WAIT_UNSET (sc_uint<4>)10

void ProcessLines::processLinePair()
{
// CheckInside interface
sc_logic ins_data_set;

// CheckIntersect interface
sc_logic int_data_set;

// ComputeT0 interface
sc_logic t0_data_set;

// ComputeU0 interface
sc_logic u0_data_set;

// ComputeCoordinate interface
sc_logic cc_data_set;

// ProcessLines interface
sc_int<9>  p1X;
sc_int<9>  p1Y;
sc_int<9>  p2X;
sc_int<9>  p2Y;
sc_int<9>  p3X;
sc_int<9>  p3Y;
sc_int<9>  p4X;
sc_int<9>  p4Y;
sc_logic   data_set;
sc_logic   result_set;
```

```
  sc_uint<2> result;
  sc_uint<2> inside1, inside2;
  sc_uint<2> intersect;
  float t0;
  float u0;
sc_int<9> newX1, newY1;
sc_uint<4> p_state = STATE_WAIT_SET;

while(true)
{
if (p_state == STATE_WAIT_SET)
{
result_set = '0';
result_sets = result_set;
ins_data_set = '0';
int_data_set = '0';
t0_data_set = '0';
u0_data_set = '0';
cc_data_set = '0';
ins_data_sets = ins_data_set;
int_data_sets = int_data_set;
t0_data_sets = t0_data_set;
u0_data_sets = u0_data_set;
cc_data_sets = cc_data_set;

data_set = data_sets;
if (data_set == '1')
{
p1X = p1Xs;
p1Y = p1Ys;
p2X = p2Xs;
p2Y = p2Ys;
p3X = p3Xs;
p3Y = p3Ys;
p4X = p4Xs;
p4Y = p4Ys;

newX1 = 255;
newY1 = 255;
p_state = STATE_INSIDE_1;
}
}
else if (p_state == STATE_INSIDE_1)
{
ins_p1Xs = p3X;
ins_p1Ys = p3Y;
ins_p2Xs = p4X;
ins_p2Ys = p4Y;
ins_p3Xs = p1X;
ins_p3Ys = p1Y;
ins_data_set = '1';
ins_data_sets = ins_data_set;
```

```
wait();
wait_until(ins_result_sets.delayed() == true);
inside1 = ins_result_outs;
ins_data_set = '0';
ins_data_sets = ins_data_set;
p_state = STATE_INSIDE_2;
}
else if (p_state == STATE_INSIDE_2)
{
ins_p1Xs = p3X;
ins_p1Ys = p3Y;
ins_p2Xs = p4X;
ins_p2Ys = p4Y;
ins_p3Xs = p2X;
ins_p3Ys = p2Y;
ins_data_set = '1';
ins_data_sets = ins_data_set;
wait();
wait_until(ins_result_sets.delayed() == true);
inside2 = ins_result_outs;
ins_data_set = '0';
ins_data_sets = ins_data_set;
p_state = STATE_T0;
}
else if (p_state == STATE_T0)
{
t0_p1Xs = p1X;
t0_p1Ys = p1Y;
t0_p2Xs = p2X;
t0_p2Ys = p2Y;
t0_p3Xs = p3X;
t0_p3Ys = p3Y;
t0_p4Xs = p4X;
t0_p4Ys = p4Y;
t0_data_set = '1';
t0_data_sets = t0_data_set;
wait();
wait_until(t0_result_sets.delayed() == true);
t0 = t0_result_outs;
t0_data_set = '0';
t0_data_sets = t0_data_set;
p_state = STATE_U0;
}
else if (p_state == STATE_U0)
{
u0_p1Xs = p1X;
u0_p1Ys = p1Y;
u0_p2Xs = p2X;
u0_p2Ys = p2Y;
u0_p3Xs = p3X;
u0_p3Ys = p3Y;
u0_p4Xs = p4X;
u0_p4Ys = p4Y;
```

```
u0_t0s = t0;
u0_data_set = '1';
u0_data_sets = u0_data_set;
wait();
wait_until(u0_result_sets.delayed() == true);
u0 = u0_result_outs;
u0_data_set = '0';
u0_data_sets = u0_data_set;
p_state = STATE_INTERSECT;
}
else if (p_state == STATE_INTERSECT)
{
int_p1Xs = p1X;
int_p1Ys = p1Y;
int_p2Xs = p2X;
int_p2Ys = p2Y;
int_p3Xs = p3X;
int_p3Ys = p3Y;
int_p4Xs = p4X;
int_p4Ys = p4Y;
int_t0s = t0;
int_u0s = u0;
int_data_set = '1';
int_data_sets = int_data_set;
wait();
wait_until(int_result_sets.delayed() == true);
intersect = int_result_outs;
int_data_set = '0';
int_data_sets = int_data_set;


if (inside1 == OUTSIDE && inside2 == INSIDE && intersect == INTERSECT)
result = ADD_NEW_POINT_ENTER;
else if (inside1 == INSIDE && inside2 == OUTSIDE &&
                                                    intersect == INTERSECT)
      result = ADD_NEW_POINT_LEAVE;
else
result = ADD_NONE;

if (result != ADD_NONE)
p_state = STATE_NEW_X;
else
p_state = STATE_SEND_DATA;
}
else if(p_state == STATE_NEW_X)
{
cc_v1s = p1X;
cc_v2s = p2X;
cc_t0s = t0;
cc_data_set = '1';
cc_data_sets = cc_data_set;
wait();
wait_until(cc_result_sets.delayed() == true);
```

```
newX1 = cc_result_outs;
cc_data_set = '0';
cc_data_sets = cc_data_set;
p_state = STATE_NEW_Y;
}
else if (p_state == STATE_NEW_Y)
{
cc_v1s = p1Y;
cc_v2s = p2Y;
cc_t0s = t0;
cc_data_set = '1';
cc_data_sets = cc_data_set;
wait();
wait_until(cc_result_sets.delayed() == true);
newY1 = cc_result_outs;
cc_data_set = '0';
cc_data_sets = cc_data_set;
p_state = STATE_SEND_DATA;
}
else if (p_state == STATE_SEND_DATA)
{
result_outs = result;
resultX_outs = newX1;
resultY_outs = newY1;
result_set = '1';
result_sets = result_set;
p_state = STATE_WAIT_UNSET;
}
else if (p_state == STATE_WAIT_UNSET)
{
data_set = data_sets;
if (data_set == '0')
{
result_set = '0';
result_sets = result_set;
p_state = STATE_WAIT_SET;
}
}
wait();
}
}
```

```
#ifndef __TEST_DRVR_H
#define __TEST_DRVR_H

#include "scv.h"

#define STATE_WAIT_INPUT  (sc_uint<2>)1
#define STATE_HW_DISPATCH (sc_uint<2>)2
```

```
#define STATE_WAIT_OUTPUT (sc_uint<2>)3
#define CTRL_DATA_WAIT    1
#define CTRL_DATA_READY   2

SC_MODULE(Test_Drvr)
{
public:
sc_in_clk         clk;
sc_out<sc_bv<9> > p1Xs;
sc_out<sc_bv<9> > p1Ys;
sc_out<sc_bv<9> > p2Xs;
sc_out<sc_bv<9> > p2Ys;
sc_out<sc_bv<9> > p3Xs;
sc_out<sc_bv<9> > p3Ys;
sc_out<sc_bv<9> > p4Xs;
sc_out<sc_bv<9> > p4Ys;
sc_out<sc_logic > data_sets;
sc_in<sc_bv<2> >  result_outs;
sc_in<sc_bv<9> > resultX_outs;
sc_in<sc_bv<9> > resultY_outs;
sc_in<sc_logic >  result_sets;

struct TPointStruct
{
sc_int<9> p1X;
sc_int<9> p1Y;
sc_int<9> p2X;
sc_int<9> p2Y;
sc_int<9> p3X;
sc_int<9> p3Y;
sc_int<9> p4X;
sc_int<9> p4Y;
sc_int<9> resultX;
sc_int<9> resultY;
sc_uint<2> result;
};

void runTest();
int processLinePair(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y,
                    int p4X, int p4Y, int *newX, int *newY);

SC_CTOR(Test_Drvr)
{
SC_CTHREAD(runTest, clk.pos());
}
};

#endif
```

```
#include "Test_Drvr.h"

static Test_Drvr::TPointStruct line_data;
static int c_state = CTRL_DATA_WAIT;
static sc_mutex r_mutex;

void Test_Drvr::runTest()
{
sc_uint<2> p_state;
bool mutex_set = false;
sc_logic data_set;

data_set = '0';
data_sets = data_set;

p_state = STATE_WAIT_INPUT;
line_data.result = -3;

wait();
while(true)
{
if (p_state == STATE_WAIT_INPUT)
{
if (!mutex_set)
{
r_mutex.lock();
mutex_set = true;
}
if (c_state == CTRL_DATA_READY)
p_state = STATE_HW_DISPATCH;
}
else if (p_state == STATE_HW_DISPATCH)
{
p1Xs = line_data.p1X;
p1Ys = line_data.p1Y;
p2Xs = line_data.p2X;
p2Ys = line_data.p2Y;
p3Xs = line_data.p3X;
p3Ys = line_data.p3Y;
p4Xs = line_data.p4X;
p4Ys = line_data.p4Y;
data_set = '1';
data_sets = data_set;
wait();
wait_until(result_sets.delayed() == true);
line_data.result = result_outs;
line_data.resultX = resultX_outs;
line_data.resultY = resultY_outs;
data_set = '0';
data_sets = data_set;
if (mutex_set)
{
```

```
r_mutex.unlock();
mutex_set = false;
}
p_state = STATE_WAIT_OUTPUT;
}
else if (p_state == STATE_WAIT_OUTPUT)
{
if (c_state == CTRL_DATA_WAIT)
p_state = STATE_WAIT_INPUT;
}
wait();
}
}

int Test_Drvr::processLinePair(int p1X, int p1Y, int p2X, int p2Y, int p3X, int p3Y, int p4X, int p4Y, i
{
int result;

wait();
line_data.p1X = p1X;
line_data.p1Y = p1Y;
line_data.p2X = p2X;
line_data.p2Y = p2Y;
line_data.p3X = p3X;
line_data.p3Y = p3Y;
line_data.p4X = p4X;
line_data.p4Y = p4Y;
c_state = CTRL_DATA_READY;
r_mutex.lock();
result = (int)line_data.result;
*newX = (int)line_data.resultX;
*newY = (int)line_data.resultY;
c_state = CTRL_DATA_WAIT;
r_mutex.unlock();
return result;
}



#ifndef __PROCESSPOLYGONS_H
#define __PROCESSPOLYGONS_H

#include "ClipConstants.h"
class ProcessPolygons
{
public:

int processPolygons(int *poly1, int *poly2, int count1, int count2,
                    int *resultPoly);
ProcessPolygons();
~ProcessPolygons();

private:
```

```
void findIntersections(int *poly1, int *poly2, int *count1, int *count2,
                    int *resultPoly, int *countR);
int findEnterPoint(int *poly, int count);
void swapIntegers(int *v1, int *v2);
void swapPointers(int **p1, int **p2);
void insertPoint(int x, int y, int *poly, int psn, int *pcount, int status);
void insertResultPoint(int x, int y, int *poly, int psn, int *pcount);
bool addPoint(int x, int y, int *poly, int *pcount);
void setStatus(int *poly, int psn, int status);
int getStatus(int *poly,int psn);
void insertInsidePoints(int *poly1, int *poly2, int max1, int max2,
                        int *resultPoly, int *maxR);
int resultPolygonCount(int *poly, int count);
void listPolygon(int *poly, int count);
void listPolygonR(int *poly, int count);
bool pointExistsOutput(int *poly, int x, int y, int pcount);
};
#endif
```

```
#include "ProcessPolygons.h"
#include "Test_Drvr.h"

extern Test_Drvr *tDrvr;

void printData1(int *, int);
void printData2(int *, int);

int ProcessPolygons::processPolygons(int *poly1, int *poly2, int count1,
                                  int count2, int *resultPoly)
{
int max1 = count1, max2 = count2;
int resCount = 0;
int resultCount = 0;
findIntersections(poly1, poly2, &max1, &max2, resultPoly, &resCount);
//printData1(poly1, max1);
//printData1(poly2, max2);
//printData2(resultPoly, resCount);
insertInsidePoints(poly1, poly2, max1, max2, resultPoly, &resCount);
//printData2(resultPoly, resCount);
resultCount =  resultPolygonCount(resultPoly, resCount);

return resultCount;
}

void ProcessPolygons::findIntersections(int *poly1, int *poly2, int *count1,
                                    int *count2, int *resultPoly, int *countR)
```

```
{
int resultCount = 0;
int resCount = 0;
int result;
int lastResult = ADD_NONE;
int lastCount1 = *count1 - 1, lastCount2 = *count2 - 1;
int max1 = *count1, max2 = *count2;
int iterationCount;
int *lPoly1 = poly1, *lPoly2 = poly2;
bool swapFlag = false;
bool intersectFlag = true;
int newX =  255, newY =  255;
int i, j;
int p1X, p1Y, p2X, p2Y, p3X, p3Y, p4X, p4Y;
int startX = 255, startY = 255;


while (intersectFlag)
{
intersectFlag = false;
iterationCount = 0;
i = (lastCount1 + 1) % max1;
while (i != lastCount1 && iterationCount <= max1)
{
{
p1X = *(lPoly1 + (3 * i));
p1Y = *(lPoly1 + (3 * i) + 1);
p2X = *(lPoly1 + (3 * ((i + 1) % max1)));
p2Y = *(lPoly1 + (3 * ((i + 1) % max1)) + 1);
j = (lastCount2 + 1) % max2;
while (j != lastCount2)
{

newX = 255;
newY = 255;

{
p3X = *(lPoly2 + (3 * j));
p3Y = *(lPoly2 + (3 * j) + 1);
p4X = *(lPoly2 + (3 * ((j + 1) % max2)));
p4Y = *(lPoly2 + (3 * ((j + 1) % max2)) + 1);

result = tDrvr->processLinePair(p1X, p1Y, p2X, p2Y, p3X,
                                        p3Y, p4X, p4Y, &newX,
                                        &newY);
if ((result != (unsigned)ADD_NONE) &&
    (pointExistsOutput(resultPoly, newX, newY, resCount)))
{
result = ADD_NONE;
newX = 255;
newY = 255;
}
```

```
if (result == (int)ADD_NEW_POINT_ENTER)
{
intersectFlag = true;
lastResult = ADD_NEW_POINT_ENTER;
addPoint(newX, newY, resultPoly, &resCount);
insertPoint(newX, newY, lPoly1, i, &max1, INTERSECT_ENTER);
insertPoint(newX, newY, lPoly2, j, &max2, INTERSECT_LEAVE);
resultCount++;
lastCount1 = i;
lastCount2 = j;
startX = newX;
startY = newY;
break;
}
else if (result == (int)ADD_NEW_POINT_LEAVE)
{
intersectFlag = true;
lastResult = ADD_NEW_POINT_LEAVE;
addPoint(newX, newY, resultPoly, &resCount);
swapFlag = true;
insertPoint(newX, newY, lPoly1, i, &max1, INTERSECT_LEAVE);
insertPoint(newX, newY, lPoly2, j, &max2, INTERSECT_ENTER);
lastCount1 = i;
lastCount2 = j;
break;
}
else
{
if (p2X == p4X && p2Y == p4Y && p2X == startX && p2Y == startY )
{
resCount++;
startX = 255;
startY = 255;
}
}
}

j = (j + 1) % max2;
}
if (swapFlag)
break;
}
iterationCount = iterationCount + 1;
i = (i + 1) % max1;
}
if (swapFlag)
{
swapFlag = false;
swapPointers(&lPoly1, &lPoly2);
swapIntegers(&max1, &max2);
swapIntegers(&lastCount1, &lastCount2);
lastResult = ADD_NEW_POINT_ENTER;
}
```

```
}

if (lPoly1 == poly1)
{
*count1 = max1;
*count2 = max2;
}
else
{
*count1 = max2;
*count2 = max1;
}
*countR = resCount;
}

void ProcessPolygons::insertInsidePoints(int *poly1, int *poly2, int max1,
                                         int max2, int *resultPoly, int *maxR)
{
int enterX, enterY;
int i, k;

while ((i = findEnterPoint(poly1, max1)) != max1 + 1)
{
setStatus(poly1, i, INTERSECT_DONE);
enterX = *(poly1 + (3 * i));
enterY = *(poly1 + (3 * i) + 1);
i = (i + 1) % max1;
k = 0;
for (k = 0; *(resultPoly + (2 * k)) != enterX ||
            *(resultPoly + (2 * k) + 1) != enterY; k++);
while (*(poly1 + (3 * i) + 2) != (unsigned char)INTERSECT_LEAVE)
{
insertResultPoint(*(poly1 + (3 * i)),  *(poly1 + (3 * i) + 1),
                  resultPoly, k, maxR);
k = k + 1;
i = (i + 1) % max1;
}
}

while ((i = findEnterPoint(poly2, max2)) != max2 + 1)
{
setStatus(poly2, i, INTERSECT_DONE);
enterX = *(poly2 + (3 * i));
enterY = *(poly2 + (3 * i) + 1);
i = (i + 1) % max2;
k = 0;
for (k = 0; *(resultPoly + (2 * k)) != enterX ||
            *(resultPoly + (2 * k) + 1) != enterY; k++);
while (*(poly2 + (3 * i) + 2) != (unsigned char)INTERSECT_LEAVE)
{
insertResultPoint(*(poly2 + (3 * i)),  *(poly2 + (3 * i) + 1),
                  resultPoly, k, maxR);
k = k + 1;
```

```
i = (i + 1) % max2;
}
}
}

int ProcessPolygons::findEnterPoint(int *poly, int count)
{
int i;

for (i = 0; i < count; i++)
if (*(poly + (3 * i) + 2) == (unsigned char)INTERSECT_ENTER)
return i;
return count + 1;
}

int ProcessPolygons::resultPolygonCount(int *poly, int pcount)
{
int i;
int resultCount = 0;

for (i = 0; i < pcount; i++)
if ((*(poly + (2 * i)) != 255) && (*(poly + (2 * (i + 1))) == 255))
resultCount = resultCount + 1;

return resultCount;
}

void ProcessPolygons::swapIntegers(int *v1, int *v2)
{
int t;

t = *v1;
*v1 = *v2;
*v2 = t;
}

void ProcessPolygons::swapPointers(int **p1, int **p2)
{
int *t;

t = *p1;
*p1 = *p2;
*p2 = t;
}


void ProcessPolygons::insertPoint(int x, int y,int *poly,int psn, int *pcount,
                                  int status)
{
int start = psn + 1;
int end = *pcount;
int i;
bool found = false;
```

```
for (i = 0; i < end; i++)
if ((*(poly + (3 * i)) == x) && (*(poly + (3 * i) + 1) == y))
found = true;

if (!found)
{
for (i = end - 1; i >= start; i--)
{
*(poly + (3 * (i + 1))) = *(poly + (3 * i));
*(poly + (3 * (i + 1)) + 1) = *(poly + (3 * i) + 1);
*(poly + (3 * (i + 1)) + 2) = *(poly + (3 * i) + 2);
}
*(poly + (3 * start)) = x;
*(poly + (3 * start) + 1) = y;
setStatus( poly, start, status);
*pcount = end + 1;
}
}

void ProcessPolygons::insertResultPoint(int x, int y,int *poly,int psn,
                                  int *pcount)
{
int start = psn + 1;
int end = *pcount;
int i;
bool found = false;

for (i = 0; i < end; i++)
if ((*(poly + (2 * i)) == x) && (*(poly + (2 * i) + 1) == y))
found = true;

if (!found)
{
for (i = end - 1; i >= start; i--)
{
*(poly + (2 * (i + 1))) = *(poly + (2 * i));
*(poly + (2 * (i + 1)) + 1) = *(poly + (2 * i) + 1);
}
*(poly + (2 * start)) = x;
*(poly + (2 * start) + 1) = y;
*pcount = end + 1;
}
}



bool ProcessPolygons::addPoint(int x, int y,int *poly, int *pcount)
{
int end = *pcount;
int i;
bool found = false;
```

```
for (i = 0; i < end; i++)
if ((*(poly + (2 * i)) == x) && (*(poly + (2 * i) + 1) == y))
found = true;

if (!found)
{
*(poly + (2 * i)) = x;
*(poly + (2 * i) + 1) = y;
*pcount = i + 1;
}

return found;
}

void ProcessPolygons::setStatus(int *poly, int psn, int status)
{
*(poly + (3 * psn) + 2) = status;
}

int ProcessPolygons::getStatus(int *poly, int psn)
{
return *(poly + (3 * psn) + 2);
}

bool ProcessPolygons::pointExistsOutput(int *poly, int x, int y, int pcount)
{
bool found = false;
int i;

for (i = 0; i < pcount; i++)
if ((*(poly + (2 * i)) == x) && (*(poly + (2 * i) + 1) == y))
found = true;
return found;
}

ProcessPolygons::ProcessPolygons()
{
}


ProcessPolygons::~ProcessPolygons()
{
}

/* The following code is for debugging purposes only. */

void printData1(int *poly, int c)
{
int i;

printf("\n");
for (i = 0; i < c; i++)
printf("(%3d, %3d) %d\n", *(poly + (3 * i)), *(poly + (3 * i) + 1), *(poly + (3 * i) + 2));
```

```
}

void printData2(int *poly, int c)
{
int i;

printf("\n");
for (i = 0; i < c; i++)
printf("(%3d, %3d) \n", *(poly + (2 * i)), *(poly + (2 * i) + 1));
}
```

```
#include "scv.h"
#include "ClipConstants.h"

SC_MODULE(CheckInside)
{
sc_in_clk         clk;
sc_in<sc_bv<9> >  p1Xs;
sc_in<sc_bv<9> >  p1Ys;
sc_in<sc_bv<9> >  p2Xs;
sc_in<sc_bv<9> >  p2Ys;
sc_in<sc_bv<9> >  p3Xs;
sc_in<sc_bv<9> >  p3Ys;
sc_in<sc_logic >  data_sets;
sc_out<sc_bv<2> > result_outs;
sc_out<sc_logic > result_sets;

void checkInside();

SC_CTOR(CheckInside)
{
SC_CTHREAD(checkInside, clk.pos());
}
};
```

```
#include <iostream>
#include "CheckInside.h"

void CheckInside::checkInside()
{
sc_int<9> p1X;
sc_int<9> p1Y;
sc_int<9> p2X;
sc_int<9> p2Y;
sc_int<9> p3X;
```

```
sc_int<9> p3Y;
sc_logic  data_set;
sc_logic  result_set;

  sc_uint<2> result;
  sc_int<9> xn, xn1, yn;
  sc_int<9> xt, yt;
  sc_int<32> d, d1, d2;

result_set = '0';
result_sets = result_set;


wait();
while(true)
{

data_set = data_sets;

if (data_set == '1')
{
p1X = p1Xs;
p1Y = p1Ys;
p2X = p2Xs;
p2Y = p2Ys;
p3X = p3Xs;
p3Y = p3Ys;

xn1 = p2Y - p1Y;
xn = -xn1;
yn = p2X - p1X;
xt = p3X - p1X;
yt = p3Y - p1Y;
d1 = xn * xt;
d2 = yn * yt;
d = d1 + d2;
if (d == (sc_int<32>)0)
result = ON_LINE;
else if (d > (sc_int<32>)0)
result = OUTSIDE;
else
result = INSIDE;

result_outs = result;
result_set = '1';
result_sets = result_set;
}
else
{
result_set = '0';
result_sets = result_set;
}
wait();
```

```
}

}
```

```
#include "scv.h"

SC_MODULE(ComputeT0)
{
public:
sc_in_clk         clk;
sc_in<sc_bv<9> >  p1Xs;
sc_in<sc_bv<9> >  p1Ys;
sc_in<sc_bv<9> >  p2Xs;
sc_in<sc_bv<9> >  p2Ys;
sc_in<sc_bv<9> >  p3Xs;
sc_in<sc_bv<9> >  p3Ys;
sc_in<sc_bv<9> >  p4Xs;
sc_in<sc_bv<9> >  p4Ys;
sc_in<sc_logic >  data_sets;
sc_out<float >    result_outs;
sc_out<sc_logic > result_sets;

void computeT0();
float cvt_int_float(sc_int<32> val);

SC_CTOR(ComputeT0)
{
SC_CTHREAD(computeT0, clk.pos());
}
};
```

```
#include "ComputeT0.h"

void ComputeT0::computeT0()
{
sc_int<9>  p1X;
sc_int<9>  p1Y;
sc_int<9>  p2X;
sc_int<9>  p2Y;
sc_int<9>  p3X;
sc_int<9>  p3Y;
sc_int<9>  p4X;
```

```
sc_int<9>  p4Y;
sc_logic   data_set;
sc_logic   result_set;

  sc_int<9> t1, t2, t4, t5;
  sc_int<9> b1, b4;
  sc_int<32> t3, t6, t7;
  sc_int<32> b3, b6, b7;
  float r1, r2, r3;

result_set = '0';
result_sets = result_set;

wait();
while(true)
{
data_set = data_sets;

if (data_set == '1')
{
p1X = p1Xs;
p1Y = p1Ys;
p2X = p2Xs;
p2Y = p2Ys;
p3X = p3Xs;
p3Y = p3Ys;
p4X = p4Xs;
p4Y = p4Ys;

t1 = p3X - p1X;
t2 = p4Y - p3Y;
t3 = t1 * t2;
t4 = p3Y - p1Y;
t5 = p4X - p3X;
t6 = t4 * t5;
t7 = t3 - t6;
b1 = p2X - p1X;
b3 = b1 * t2;
b4 = p2Y - p1Y;
b6 = b4 * t5;
b7 = b3 - b6;

r2 = cvt_int_float(t7);
r3 = cvt_int_float(b7);
r1 = r2 / r3;

result_outs = r1;
result_set = '1';
result_sets = result_set;

}
else
{
```

```
result_set = '0';
result_sets = result_set;
}
wait();
}
}


float ComputeT0::cvt_int_float(sc_int<32> val)
{
float f;

f = (float)val;

return f;
}




#include "scv.h"

SC_MODULE(ComputeU0)
{
public:
sc_in_clk         clk;
sc_in<sc_bv<9> >  p1Xs;
sc_in<sc_bv<9> >  p1Ys;
sc_in<sc_bv<9> >  p2Xs;
sc_in<sc_bv<9> >  p2Ys;
sc_in<sc_bv<9> >  p3Xs;
sc_in<sc_bv<9> >  p3Ys;
sc_in<sc_bv<9> >  p4Xs;
sc_in<sc_bv<9> >  p4Ys;
sc_in<float >     t0s;
sc_in<sc_logic >  data_sets;
sc_out<float >    result_outs;
sc_out<sc_logic > result_sets;

void computeU0();
float cvt_int_float(sc_int<9> val);

SC_CTOR(ComputeU0)
{
SC_CTHREAD(computeU0, clk.pos());
}
};
```

```
#include "ComputeU0.h"

void ComputeU0::computeU0()
{
sc_int<9>  p1X;
sc_int<9>  p1Y;
sc_int<9>  p2X;
sc_int<9>  p2Y;
sc_int<9>  p3X;
sc_int<9>  p3Y;
sc_int<9>  p4X;
sc_int<9>  p4Y;
float      t0;
sc_logic   data_set;
sc_logic   result_set;

sc_int<9> b1, b2;
sc_int<9> t1;
float f1, f2, f3, f4, f5, f6, f7, f8;

result_set = '0';
result_sets = result_set;

wait();
while(true)
{
data_set = data_sets;

if (data_set == '1')
{
p1X = p1Xs;
p1Y = p1Ys;
p2X = p2Xs;
p2Y = p2Ys;
p3X = p3Xs;
p3Y = p3Ys;
p4X = p4Xs;
p4Y = p4Ys;
t0 = t0s;

b1 = p4X - p3X;
b2 = p4Y - p3Y;

if (b1 != (sc_int<9>)0)
{
t1 = p2X - p1X;
f1 = cvt_int_float(t1);
f2 = f1 * t0;
f8 = cvt_int_float(p1X);
f3 = f8 + f2;
f4 = cvt_int_float(p3X);
```

```
f5 = f3 - f4;
f6 = cvt_int_float(b1);
f7 = f5 / f6;
}
else if (b2 != (sc_int<9>)0)
{
t1 = p2Y - p1Y;
f1 = cvt_int_float(t1);
f2 = f1 * t0;
f8 = cvt_int_float(p1Y);
f3 = f8 + f2;
f4 = cvt_int_float(p3Y);
f5 = f3 - f4;
f6 = cvt_int_float(b2);
f7 = f5 / f6;
}
else
f7 = 100.0;

result_outs = f7;
result_set = '1';
result_sets = result_set;
}
else
{
result_set = '0';
result_sets = result_set;
}
wait();
}
}


float ComputeU0::cvt_int_float(sc_int<9> val)
{
float f;

f = (float)val;

return f;
}




#include "scv.h"

SC_MODULE(CheckIntersect)
{
public:
sc_in_clk         clk;
```

```
sc_in<sc_bv<9> >  p1Xs;
sc_in<sc_bv<9> >  p1Ys;
sc_in<sc_bv<9> >  p2Xs;
sc_in<sc_bv<9> >  p2Ys;
sc_in<sc_bv<9> >  p3Xs;
sc_in<sc_bv<9> >  p3Ys;
sc_in<sc_bv<9> >  p4Xs;
sc_in<sc_bv<9> >  p4Ys;
sc_in<float >     t0s;
sc_in<float >     u0s;
sc_in<sc_logic >  data_sets;
sc_out<sc_bv<2> > result_outs;
sc_out<sc_logic > result_sets;

void checkIntersect();

SC_CTOR(CheckIntersect)
{
SC_CTHREAD(checkIntersect, clk.pos());
}
};
```

```
#include "CheckIntersect.h"
#include "ClipConstants.h"

void CheckIntersect::checkIntersect()
{
sc_int<9> p1X;
sc_int<9> p1Y;
sc_int<9> p2X;
sc_int<9> p2Y;
sc_int<9> p3X;
sc_int<9> p3Y;
sc_int<9> p4X;
sc_int<9> p4Y;
float     t0;
float     u0;
sc_logic  data_set;
sc_logic  result_set;

sc_uint<2> result;
sc_int<19> d;
sc_int<19> t3, t6;;
sc_int<9> t1, t2, t4, t5;

result_set = '0';
result_sets = result_set;
```

```
wait();
while(true)
{

data_set = data_sets;

if (data_set == '1')
{
p1X  = p1Xs;
p1Y  = p1Ys;
p2X  = p2Xs;
p2Y  = p2Ys;
p3X  = p3Xs;
p3Y  = p3Ys;
p4X  = p4Xs;
p4Y  = p4Ys;
t0 = t0s;
u0 = u0s;

t1 = p2X - p1X;
t2 = p4Y - p3Y;
t3 = t1 * t2;
t4 = p2Y - p1Y;
t5 = p4X - p3X;
t6 = t4 * t5;
d = t3 - t6;

result = NO_INTERSECT;
if (d != (sc_int<19>)0)
{
if (t0 >= 0.0 && 1.0 >= t0)
{
if (0.0 <= u0 && u0 <= 1.0)
{
result = INTERSECT;
}
}
}

result_outs = result;
result_set = '1';
result_sets = result_set;
}
else
{
result_set = '1';
result_sets = result_set;
}
wait();
}
}
```

```
#include "scv.h"

SC_MODULE(ComputeCoordinate)
{
public:
sc_in_clk         clk;
sc_in<sc_bv<9> >  v1s;
sc_in<sc_bv<9> >  v2s;
sc_in<float >     t0s;
sc_in<sc_logic >  data_sets;
sc_out<sc_bv<9> > result_outs;
sc_out<sc_logic > result_sets;

void computeCoordinate();
sc_int<9> round(float val);
float cvt_int_float(sc_int<9> val);

SC_CTOR(ComputeCoordinate)
{
SC_CTHREAD(computeCoordinate, clk.pos());
}

};
```

```
#include "ComputeCoordinate.h"

void ComputeCoordinate::computeCoordinate()
{
sc_int<9> v1;
sc_int<9> v2;
float     t0;
sc_logic data_set;
sc_logic result_set;

float f1, f2;
sc_int<9> t1, t2, t3;

result_set = '0';
result_sets = result_set;

wait();
while(true)
{
data_set = data_sets;
```

```
if (data_set == '1')
{
v1 = v1s;
v2 = v2s;
t0 = t0s;

t1 = v2 - v1;
f1 = cvt_int_float(t1);
f2 = f1 * t0;
t2 = round(f2);
t3 = v1 + t2;

result_outs = t3;
result_set = '1';
result_sets = result_set;
}
else
{
result_set = '1';
result_sets = result_set;
}
wait();
}
}

sc_int<9> ComputeCoordinate::round(float val)
{
sc_int<9> t1;
float f1;

t1 = (sc_int<9>)val;
f1 = val - t1;
if (f1 >= 0.5)
t1 = t1 + 1;
return t1;
}

float ComputeCoordinate::cvt_int_float(sc_int<9> val)
{
float f;

f = (float)val;

return f;
}
```