Conversion of Cray pointers to Fortran 90 pointers in a Ray tracing application

Supervisor: Prof. Sven-Erik Sandström
Acknowledgement

All praises to Almighty Allah, whose enormous blessings give us strength and make us able to complete this thesis. We would like to thank our supervisor Prof. Sven-Erik Sandström for his kind support, assistance and guidance throughout our thesis. His motivation was a source of inspiration to our thesis. Finally, all blessings upon our beloved parents and families whose prayers and moral support always motivated us to complete our studies.
Abstract

This report is a documentation of techniques involved in the conversion of Cray pointers to Fortran 90 pointers in a ray-tracing application. A CAD object is generated and used for ray-tracing and the output is rendered on a pixel screen. Several implementations of ray-tracing were compared. A number of simulations were made to illustrate the properties of the code raypol. Spatial sub-division methods like Grid structures and BSP are implemented with pointers. Programming aspects relating to the use of pointers in the Fortran language are discussed. Portability is improved by conversion to Fortran 90 pointers and the use of other Fortran 90 features.
Contents

1 Introduction 6
   1.1 Thesis objective .......................................... 6
   1.2 History .................................................. 6
   1.3 Thesis organization ...................................... 7

2 Ray tracing implementation 8
   2.1 Creation of CAD objects .................................. 8
   2.2 Objects .................................................. 8
   2.3 Ray tracing .............................................. 9
   2.4 Calculation of the ray intersection ....................... 9
      2.4.1 Ray-sphere intersection ............................. 10
   2.5 The ray tracing algorithm ................................ 11
   2.6 Spatial sub-division in ray tracing ....................... 11
      2.6.1 The Grid Partitioning method ....................... 11
      2.6.2 The Binary Space Partitioning method ............... 11
   2.7 Rendering based on ray tracing .......................... 11

3 Pointers 14
   3.1 The declaration of Cray pointers ......................... 14
   3.2 Features of the Cray pointers ............................ 15
   3.3 Declaration of the Fortran 90 pointers ................... 15
   3.4 The pointer assignment .................................. 16
   3.5 The target declaration ................................... 17
   3.6 Dynamic targets ......................................... 17
   3.7 Basic differences between the Cray pointer and the Fortran 90 pointer ........................................... 18
   3.8 Restrictions of the Cray pointers ....................... 19
   3.9 Advantages of the Fortran 90 pointers ................... 19

4 Derived data types 20
   4.1 Basics .................................................. 20
   4.2 Derived data type assignment ............................ 20
   4.3 Derived data types and the pointer components .......... 21
   4.4 Types with f90 pointers for linked list/BSP tree .......... 21
   4.5 A linked list example ................................... 22

5 Modules 25
   5.1 Background ............................................... 25
   5.2 The structure of a Module ................................ 25
5.3 Specific modules ........................................... 26
  5.3.1 Module create_list_and_bounds .................... 26
  5.3.2 Module grid_library .................................. 26
  5.3.3 Module binary_tree_library ......................... 26

6 BSP ......................................................... 27
  6.1 Introduction to the BSP tree ........................... 27
  6.2 Generating a BSP tree .................................. 27
  6.3 Algorithm for BSP tree .................................. 28
      6.3.1 Schematic algorithm ............................... 29

7 Concluding remarks ...................................... 30
  7.1 Problems encountered in the conversion ............... 30
  7.2 Comparison of the implementation of linked lists ... 30

8 Conclusion ................................................ 32

Appendix A The program Polgen ............................ 35

Appendix B Examples ........................................ 40
  B.1 Linked list .............................................. 40
  B.2 The BSP tree ............................................. 41
# List of Figures

1. Rendering of a CAD sphere.  
2. Conesphere generated from the dxf format.  
4. The representation of a conesphere on the pixel screen.  
5. Presentation of the conesphere observed from a different angle.  
7. Initialization of the list.  
8. Initialization of a new cell.  
10. General structure of the list.  
11. Creation of a BSP tree.
1 Introduction

1.1 Thesis objective

This thesis describes the conversion of the Cray pointer facility to standard Fortran 90 pointers. Fortran 90 pointers are more compatible with Fortran 90 data facilities and array processing. The Cray pointers are adapted to Fortran 77 and Fortran 90 contains all functionalities of Fortran 77. For conversion to Fortran 90 there is the advantage that the GNU gfortran compiler allows Cray pointers and the translation can then be made in steps.

1.2 History

Fortran is still one of the dominating languages in scientific and technical applications. It was developed by IBM under the supervision of John Backus, and was gradually extended to include functions, subroutines and common blocks. A pointer is a data type which points to a value in a memory location, typically by using its address. For repeated operations on tree structures, tables and strings, the use of pointers improves performance.

Since there was no pointer facility available in Fortran 77, Cray pointers were added to remedy this shortcoming. Pointers access memory by dynamically associating variables and arrays to a specific location in a block of memory. Cray pointers use the mallocate statement to reserve memory dynamically. Cray pointers are known as non-standard pointers.

Fortran 90 pointers are introduced as the standard form of pointers in Fortran. Unlike pointers in other programming languages, the Fortran 90 pointers are specific data types known as attributes to other data types. A variable with the pointer attribute can be used as a regular variable in various ways. Fortran 90 pointers are not memory addresses as in other programming languages. Variables which are to be used as a pointer are given the attribute POINTER and all the variables that are pointed to are given the attribute TARGET. This structure increases reliability.
1.3 Thesis organization

Chapter 1 describes the objective and gives a brief history of pointers. In chapter 2, we discuss the creation of the CAD objects, acceleration structures for the ray tracing and the outputs generated by the ray tracer. The declaration, initialization and assignment of both the Cray pointers and the Fortran 90 pointers are discussed in chapter 3. Chapter 4 deals with derived data types and the linked list in Fortran 90. The creation and the organization of modules in Fortran 90 are discussed in chapter 5. Generation and implementation of the BSP tree are covered in chapter 6. Advantages and restrictions of the two pointer facilities are compared in chapter 7 and chapter 8 concludes the presentation in this thesis.
2 Ray tracing implementation

2.1 Creation of CAD objects

A Fortran code is used to generate CAD objects for ray tracing. The generated objects are also stored in the standard dxf format and can be viewed and rendered by means of standard software.

2.2 Objects

The Fortran program POLGEN generates the objects and creates a dxf file and a polygon file containing the vertices of flat facets. The standard objects are sphere, conesphere, cylinder and spheroid and they are selected by setting logical variables. To create a particular object, one sets the corresponding logical variables TRUE or FALSE in the program POLGEN. The code generates a file ‘ut.pol’ which contains geometry data for the ray tracing. POLGEN is listed in appendix A. Figure 1 is a rendering of a sphere in the dxf format, produced with free software [1].

![Figure 1: Rendering of a CAD sphere.](image)
2.3 Ray tracing

Realistic images can be produced with the ray tracing method. It is a simple and powerful approach for image synthesis. An elementary implementation of ray tracing is slow and it is therefore often accelerated by means of grid-structures or BSP (Binary Space Partitioning). Images are produced by tracing a ray from each pixel in the image plane. The rays are tested against the objects in the scene to determine if they intersect with the object [2].

Figure 3 illustrates the simple ray tracing process. The multiple rays from single point source are sent through a pixel screen and hit the object behind the screen as shown in Figure 3. Whereas the ray tracer implemented uses parallel rays which corresponds to a source which is far away from the object.

2.4 Calculation of the ray intersection

The ray-object intersection point is identified by means of simple calculations. A spherical surface and a ray in parametric form illustrates the procedure [3].
Ray-sphere intersection

Let a ray be represented by:

\[ q(t) = m + td. \] (1)

A sphere with center \( c = (x_c, y_c, z_c) \) and radius \( r \) is given by,

\[ (x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - r^2 = 0. \] (2)

The equation can be written in vector form as,

\[ (q - c) \cdot (q - c) - r^2 = 0. \] (3)

The intersection is given by the equation,

\[ (m + td - c) \cdot (m + td - c) - r^2 = 0. \] (4)

There parameter \( t \) is obtained from a second order equation,

\[ at^2 + bt + c = 0. \] (5)

with the solution,

\[ t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \] (6)

If the discriminant is negative, there is no intersection. If the discriminant is positive, there are two solutions, one for the ray entering the sphere and one for the ray leaving the sphere. A zero discriminant corresponds to a grazing ray that touches the sphere at one point.
2.5 The ray tracing algorithm

The code used for the ray tracing consists of several modules and subroutines. The Polgen.f generates the polygon file and then the Readpol.f reads the polygon file (.pol) and splits the polygons into triangular facets. To accelerate the ray tracing, either a grid structure or a BSP tree with either object or screen rotation is used. The command file (.cmd) is read and checked in Syntax.f. In order to produce a high quality representation of the object, one needs a high resolution for the pixel screen, in addition to an accurate CAD description of the object itself.

2.6 Spatial sub-division in ray tracing

To reduce the computation time, acceleration structures are introduced. The intersection calculation is the most time consuming part of the ray tracing and therefore spatial sub-division schemes like Grid structures and BSP have been devised to handle this problem.

2.6.1 The Grid Partitioning method

This method divides the volume containing the object into boxes. These boxes will then contain a number of triangular facets. This arrangement speeds up the ray-object intersection test. The grid is an effective way of administrating a large number of triangular facets. Only when a ray hits a non empty box the triangular facets in that box are tested for intersection. The process of determining the intersections continues until all pixels on the pixel screen are defined [4][5][6].

2.6.2 The Binary Space Partitioning method

A Binary Space Partitioning (BSP) tree can be used as an acceleration device for the ray tracing. This type of sub-division provides a speed-up similar to that of the grid. A discussion of the BSP tree is presented in chapter 6 [3].

2.7 Rendering based on ray tracing

The pixel-facet map produced by the ray tracing can be used to enhance the graphical presentation of the object. This is referred to as rendering and includes lighting, reflections and shadowing. The object is separated into two zones, the lit zone and the shadow zone. The pixels corresponding to the lit zone could be presented in simple RGB colors based on the angle between the surface normal and the incident light. The shadow zone pixels
are set to black. In the visualization, both the direction of the lighting and the direction of the pixel screen are essential. For convex objects there are no multiple reflections. The rendered image in Figure 4 has lighting from the left and the object is viewed from above. Figures 5 and 6 illustrate rotation of the pixel screen.

Figure 4: The representation of a conesphere on the pixel screen.

Figure 5: Presentation of the conesphere observed from a different angle.
Figure 6: Side view of a conisphere.
3 Pointers

3.1 The declaration of Cray pointers

The syntax for the declaration of a Cray pointer is:

\[
\text{Pointer} :: (\text{pointer\_name}, \text{pointee/traget}), \\
(\text{pointer\_name}, \text{pointee}(\text{dimension}))
\]

Example

\[
\begin{align*}
\text{Pointer}(& PX, X) , (PY, Y) \\
\text{Pointer}(& PA, A(10)) , (PB, B(20)) \\
\text{Pointer}(& PC, C(0 : IMAX, 0 : JMAX)) \\
\text{REAL} \ & A, B, C, X, Y
\end{align*}
\]

For Cray pointers there are two names associated with the pointer, the name of the pointer and the name of the target/pointee. Initially the pointer cannot have any value, which means that no space is set aside for the targets (A, B, C, X or Y). The pointer must be assigned to the target for the initial address. The initial address of a target is obtained from the pointer when the target is referenced. The pointer is a type of integer and only integer values are assigned to it. The values could later be used as absolute addresses [7].

Example

\[
\begin{align*}
PA &= 0 \\
PX &= 1024
\end{align*}
\]

Integer arithmetic operations can be performed on a pointer since it corresponds to an integer variable.

Example

\[
\begin{align*}
\text{INTEGER} \ & J \\
PX &= PY + 100 \\
J &= PX
\end{align*}
\]

An intrinsic function LOC returns the address of the variable, and can be used to initialize or reset the pointer.
Example

\[ \text{REAL Pool}(100) \]
\[ PA = LOC(Pool) + 10 \]

The only way to nullify a Cray pointer is to set it to zero, indicating that the pointer is not in use.

### 3.2 Features of the Cray pointers

- No address is assigned to the Cray pointee until the Cray pointer has been given a value.

- A Cray pointer cannot be pointed to by another Cray pointer or by a Fortran 90 pointer.

- A Cray pointer cannot be a component of a derived data type and cannot be declared to any other data type.

- A Cray pointer cannot appear in the Parameter statement or in any type declaration statement in which the parameter attributes are included, because the data statement has restrictions on the Cray pointees [8].

### 3.3 Declaration of the Fortran 90 pointers

In Fortran 90, the pointers are the attributes of an object. There are two ways to declare these attributes:

1. Attribute oriented declaration (in a separate statement).

2. Entity oriented declaration (in a type statement).

In general, the entity oriented declaration is preferred because all the attributes of an object are specified in a single statement. The pointer in the attribute oriented declaration can be declared as:
Pointer :: pointer_name(Dimension), pointer_name(Dimension)

Example

REAL A, B, C, X, Y
Pointer X, Y
Pointer A, B
Dimension A(:,), B(:,)
Pointer C(:,;)

In the entity oriented declaration, the pointer can be declared as:

Type, Attributes :: entity, entity
Type, Attributes, Dimension(:,::) :: entity, entity

Example

REAL, Pointer :: A, B
REAL, Pointer, Dimension(:,::) :: X(10), Y(20)
REAL, Pointer, :: Z(:,;)

In both the forms defined above, it is legal to define the dimensions either in the attribute list or with the entities. No space is set aside for the pointers A, B, X, Y or Z even after they are declared, or we can say that the association status of the pointer is undefined. Association can be achieved with the pointer assignment statement (⇒) that allocates space for the target. Absolute addresses cannot be assigned to Fortran 90 pointers. They are descriptors having the information on the type and rank of that pointer. No arithmetic operations can be performed on the pointer.

3.4 The pointer assignment

The pointer assignment statement (⇒) is used for pointing the pointer to its target.

Pointer ⇒ Target
Example

\begin{verbatim}
REAL Target :: Pool(100)
A ⇒ Pool (31)
Y ⇒ X
\end{verbatim}

In the first pointer assignment statement, A is assigned to Pool(31) which means that A is the alias of the thirty first element of Pool. In the second statement Y is given the same target as X.

### 3.5 The target declaration

The Target attribute is provided merely to aid optimization. Any non pointer object which is not declared as a target can be referred to by its original declared name. There is an attribute-oriented declaration for the Target attribute as well. The syntax for the target is as:

\begin{verbatim}
Type, Target :: target_name(Dimensions),
target_name(Dimensions)
\end{verbatim}

Example

\begin{verbatim}
REAL, Target :: Pool(100)
\end{verbatim}

### 3.6 Dynamic targets

By allocation, targets for the pointers can also be created dynamically. By using the ALLOCATE statement we can reserve space for the target of a pointer. In that case, the pointer may not be an alias of the other variable, it is a reference to an unnamed part of the storage.

Example

\begin{verbatim}
ALLOCATE (ptr1, Stat = ierr)
ALLOCATE (ptr2(4 * i, 2 * j - 1), Stat = ierr)
\end{verbatim}

The above statement is used to allocate space for both pointers; in the first statement the space is allocated for a single real and in the second for a rank 2 real array. The targets of the pointer PTR1 and PTR2 are created automatically for these objects.
In the ALLOCATE statement, the STAT= status specifier should also be used. A zero value of STAT means that the allocation is done successfully while a positive value means that an error occurred that inhibited allocation. There should be only one object per allocate statement. If a problem occurs, it is easy to diagnose what object that causes the problem. If allocated space is no longer needed during execution it should be deallocated in order to save space. The save attribute must be used if the allocated space is to be retained between the procedures calls. If a pointer is already associated then it is not an error to reallocate it. By doing this, links from the old targets will automatically be broken and new targets are installed instead [9].

In Fortran 90, the Nullify statement is used to set a pointer of any rank and type to point to no object. A logical function Associated is provided to check the status of the pointer.

Fortran 90 has two essential attributes, Pointer and Target.

3.7 Basic differences between the Cray pointer and the Fortran 90 pointer

- The Cray pointers hold the absolute address of the pointee and arithmetic operations are usually possible. The Fortran 90 pointers address the objects sequentially and use association for this. Arithmetic operations on the pointers are not possible.

- Cray pointers can be used to construct linked data structures and supports dynamic memory allocation. The Fortran 90 pointers can be used for both the dynamic memory allocation and the linked data structures. If only dynamic allocation is needed then it is better to use allocatable objects [10].

- To point to no object, the Fortran 90 pointer can be set with the NULLIFY statement. For Cray pointers this is done by setting the pointer to zero.

- The compiler assumes that the Cray pointer target does not share memory with other variables.
3.8 Restrictions of the Cray pointers

- A Cray pointer cannot be a pointee or target, and it cannot be pointed to by a f90 or Cray pointer.
- A Cray pointer variable cannot be declared to be of any other data type.
- A Cray pointer cannot appear in declaration statement or in a Parameter statement which has Parameter attribute.
- A Cray pointer cannot be a structure component.

3.9 Advantages of the Fortran 90 pointers

- In general Fortran 90 pointers make it easier to read and maintain the code.
- Fortran 90 pointers can point to a part of an array since they are descriptors with stride information and bounds.
- It is possible to create dangling pointers with no target and targets with no related pointer.
- Fortran 90 pointers are attributes which can be used as the object of any user-defined data types [7].
4 Derived data types

4.1 Basics

It is often advantageous to encode some objects in terms of comprehensive structures.

Example

- Address: name, number, street.
- Coordinates: (x,y,z)

Fortran 90 allows these compound elements to be defined as derived data types. In other languages these derived type elements are known as structures and records. With careful design, derived types make the program simple, more maintainable and more robust. It is a complex process to choose an efficient data structure.

Data types can be defined with the derived type statement.

```
Type Sphere
    REAL :: x, y, z
    REAL :: radius
End Type Sphere
```

The type sphere has four REAL components x, y, z and radius. Objects of the type sphere can be used in the same manner as the objects of the intrinsic type, for example as procedure arguments and in assignment statements.

4.2 Derived data type assignment

Values can be assigned to the elements of the derived type by the following two methods:

- Element by element
- As an object

The % operator is used to assign the values to each specific element of the derived type in a piecewise fashion. In order to select the specific element, it
is necessary that one knows the exact name that is given to the element when the derived type was initially declared. The object ball is of type sphere, and we know that ball has four elements, center coordinates x, y, z and radius. When using assignment for derived types, each element is described fully.

Example

\[
\begin{align*}
\text{ball}\%x &= 1.0 \\
\text{ball}\%y &= 2.0 \\
\text{ball}\%z &= 3.0 \\
\text{ball}\%radius &= 3.5
\end{align*}
\]

4.3 Derived data types and the pointer components

Fortran 90 does not allow an ALLOCATEABLE array as an element of a derived type, but it is possible to use f90 pointers in the derived types to construct complex data structures. A scalar, an array valued intrinsic, or a derived type can be pointed to by f90 pointers and this means that complex structures can be created and manipulated [9].

The combination of f90 pointers and derived data types allows data structures like the LINKED LIST. The pointers should be defined in the derived type to point to any other derived data type that is currently defined. It is impossible to point to a derived data type which is not yet defined and the target type must also be defined.

4.4 Types with f90 pointers for linked list/BSP tree

The type tElem is used for a linked list.

```fortran
Type tElem
  sequence
  INTEGER facnr
  Type(tElem), Pointer :: next
End Type tElem
```
Implementing a BSP tree with f90 pointers requires a more complicated type `Record` that stores children and level in the tree. The variables \( bmin(3) \) and \( bmax(3) \) are used for the boxes in the tree:

\[
Type \ Record \\
sequence \\
REAL \ast 8 \ bmin(3), bmax(3) \\
Type \ (Record), \ Pointer :: left \\
INTEGER :: \ generation \\
Type \ (Record), \ Pointer :: right \\
End \ Type \ Record
\]

### 4.5 A linked list example

The code in Appendix B.1 creates a linked list of cells. The type \( tElem \) introduced above in Section 4.4 is used in combination with the pointers \( \text{firstpek1} \) and \( \text{list1} \).

\[
Type \ (tElem), \ Pointer :: \ \text{firstpek1}, \text{list1}
\]

This pointer declaration has two f90 pointers \( \text{firstpek1} \) and \( \text{list1} \). The pointer \( \text{firstpek1} \) represents the start of the list and the list is terminated with the nullification of the pointer next (disassociated).

\[
\text{Nullify} \ (\text{firstpek1}) \\
\text{Allocate} \ (\text{firstpek1}, \ STAT = \text{AllocStat}) \\
\text{Nullify} \ (\text{firstpek1}\%next) \\
\text{firstpek1}\%facnr = 1 \\
\text{list1} \Rightarrow \text{firstpek1}
\]

![Figure 7: Initialization of the list.](image-url)
Firstpek1 is first nullified and then allocated. The next cell is nullified to produce a complete list with one cell. Initialization also involves setting firstpek1%facnr to 1 since the whole type tElem should be defined. The pointer list1 is set to point to the currently inserted point in the linked list. We know that there is no other cell in the list so the pointer list1 points to the recently inserted first cell of the list.

\[
\begin{align*}
\text{Do } & i = 2, N \\
& \text{Allocate } (\text{list1}\%\text{next}) \\
& \text{list1}\%\text{next}\%\text{facnr} = i \\
& \text{list1} \Rightarrow \text{list1}\%\text{next}
\end{align*}
\]

\text{End Do}
\text{NULLIFY } (\text{list1}\%\text{next})

Figure 8: Initialization of a new cell.

The second sequence of statements generates the rest of the list. Space is allocated dynamically in the loop and the list is terminated by nullifying the first cell outside the list.

Figure 9: Addition of the new cell.

When a new cell is created it is inserted in the list at the current position. On completion of the loop, the list has the structure shown in Figure 10 and can be traversed from the first to the last cell following the pointer links. The following code walks through the list and prints out the values of each cell. List1 is first assigned to firstpek1 that corresponds to the start of the list.

List1 is first assigned to firstpek1 that corresponds to the start of the list.
Figure 10: General structure of the list.

\[\text{list1} \Rightarrow \text{firstpek1}\]
\[\text{Do while (Associated(list1))}\]
\[\quad \text{Print*, list1}\%\text{facnr}\]
\[\quad \text{list1} \Rightarrow \text{list1}\%\text{next}\]
\[\text{End Do}\]

In this sequence of statements, the pointer list1 is the position marker in
the list. Traversal of the loop continues until the disassociated cell that
was set as a termination flag is encountered. The value of the current cell
is printed and the position in the list is shifted one step with the pointer
assignment operation. List1 is linked to list1\%next.
5 Modules

5.1 Background

Modules were introduced with the Fortran 90 standard. The use of modules must be encouraged since they are versatile and can be used to write simpler, more reusable and more reliable code. A module contains structures, pointers, operators, parameters, objects, and the procedures. One or more program units can use these definitions and specifications. Pointers must be defined within a module in order to transfer the pointers in the parameter lists of subroutines. Modules are invoked by means of the USE statement.

5.2 The structure of a Module

A module has a structure similar to that of a program unit.

Module <name of module>

    <Use [add modules]>

    Implicit none

    <Specifications>

    Contains

    <module procedure 1>
    <module procedure 2>
    ...
    <module procedure n>

End Module <name of module >

• A module has a specific name as any other program unit.
• A module can use other modules, with a USE statement before the IMPLICIT NONE statement.
• A module has a separate IMPLICIT NONE statement.
• Globally visible data can be declared before the CONTAINS statement.
• Subroutines and functions are placed after the CONTAINS statement.
• The contents of modules are also stored in coded form in separate files [11].
5.3 Specific modules

There are three modules in our code.

5.3.1 Module create_list_and_bounds

In this module the type tElem is defined inside CONTAINS so that the module can access data and recall it with the pointer firstpek1. The module contains four subroutines. Using Fortran 90 pointers, the subroutine create_list creates and stores a linked list of type tElem. A box circumscribing this set of triangles is found with the subroutine FindminBox. An overlap test is made in FillBox and triangles that overlap a certain volume are placed in newlist. The subroutine CheckTriangles finds the first out of a set of triangles that a ray encounters using the subroutine inters_triangle.

5.3.2 Module grid_library

Accelerated ray tracing by means of a grid structure is encoded in the module grid_library. The module has two data types gridbox and gridstruct, a pointer grid1 and four subroutines. Ray tracing without screen rotation is done using the subroutine Grid_rt. The triangles are rotated and this means that the grid must be regenerated for each pixel screen (picture). The subroutine Grid_rt_scrrot is used for ray tracing with screen rotation and the grid is created for the first picture only. MakeGrid makes the grid and SearchGrid searches for the closest triangle that a given ray hits.

5.3.3 Module binary_tree_library

The binary space partitioning method uses the tree structure in the module binary_tree_library. This module has a type Record, a pointer transf1 and five subroutines, four of which provide functionality similar to that of grid_library, while the fifth (CalcC) is used by SearchBSP to calculate the ray intersection with the dividing plane. BSP and ray tracing is discussed in more detail in Chapter 6.
6 BSP

6.1 Introduction to the BSP tree

Binary Space Partition trees were introduced by Fuchs, Kedem, and Naylor in 1980 [12]. The volume containing the object is sub-divided into boxes in the BSP method too. The boxes are arranged to form a binary tree by dividing each box into two subboxes. The bottom level boxes, or the leaves, contain a small number of triangular facets. This arrangement makes ray intersection testing efficient [13].

6.2 Generating a BSP tree

BSP trees are created in steps. A splitting plane/axis is chosen in order to split the box into two boxes. These boxes are then split in half along another axis which produces four sub divisions of the parent bounding box. The process is continued by repeated division in the x, y and z directions. As the boxes get smaller, they contain fewer triangular facets, and testing for ray intersection becomes faster [14].

The code in Appendix B.2 generates a simple BSP tree. The code consists of a recursive function which generates the tree structure. N defines the number of nodes of the tree. Each node has two children. The level of the tree is defined in Figure 11 and its information is stored in generation.
6.3 Algorithm for BSP tree

The algorithm for the BSP tree is explained below. The type Record is used to store all the information in the tree relating to the pointer transf1. In our code the subroutine MakeBSP is responsible for the generation of BSP tree. Memory is allocated for the incoming boxes in the tree using a pointer transf1. The level of the tree is defined. The triangles that overlap the box are known after the overlap test and are copied to the pointer newlist1. If the number of triangles that overlap the box is greater then the predefined maximum number of triangles (maxtriangles = 4) and the level of the tree is above the predefined maximum depth (maxdepth = 20), then the box is split with respect to the x,y and z axis using recursive calls until the leaf criterion is met.
6.3.1 Schematic algorithm

MakeBSP(i, trianglelist, B_min, B_max)
allocate(transf1)  ! Allocate space for incoming box in ! the binary tree.
transf1%generation = i  ! The level in the tree.

Call FillBox(trianglelist, oldlist1, B_min, B_max, tmplist, newlist1, nr_triangles)

if (nr_triangles > maxtriangles .AND. i < maxdepth)

B_maxl = B_max
B_minn = B_min
if (imod3 .EQ. 0)
    maxl = (B_max + B_min) / 2
    B_minn = B_maxl
else if (imod3 .EQ. 1)
    B_maxl = (B_max + B_min) / 2
    B_minn = B_minn
else
    B_maxl = (B_max + B_min) / 2
    B_minn = B_minn
endif
nullify(transf1%ftripek1)  ! No short list for L/R boxes.
call MakeBSP(i+1, tmplist, newlist1, tmpbox%left, transf1%left, B_min, B_maxl)
! Copy box to left node.
call MakeBSP(i+1, tmplist, newlist1, tmpbox%right, transf1%right, B_minn, B_max)
! Copy box to right node.
else
    transf1%ftripek1 => newlist1
    nullify(transf1%left)  ! L/R boxes nullified.
    nullify(transf1%right)
endif
return  ! Return pointer to transf1.
7 Concluding remarks

7.1 Problems encountered in the conversion

Initially, we tried to send f90 pointers as parameters or in common blocks but this resulted in segmentation fault. The standard method to overcome this problem is to use modules that allow transfer of pointers in the parameter list of subroutines. Modules have the restriction that they must appear in the right order so that the compiler knows what module to use.

In the testing of the code there was a compiler problem that resulted in segmentation faults. The old compiler (gfortran version 4.2.1, build 5666) does not allow multiple definitions of derived data types in a module. At the same time, pointers must be defined for the whole module in order to allow pointers in the parameter lists. The compiler seemed to compile only the first module and then produced a segmentation fault. To compensate for the shortcomings of the compiler, the modules were rearranged so that multiple declarations were avoided. The code was then compatible with the old compiler version and worked reliably.

7.2 Comparison of the implementation of linked lists

Table 1 shows the essential code for a linked list realized with f90 pointers and Cray pointers. The code with f90 pointers has a data type \textit{tElem}. This type and the f90 linked list has been used earlier. After declaration and initialization of the pointers, a linked list is generated. This implementation was discussed in detail in Section 4.5. The corresponding implementation with Cray pointers is outlined to the right in the table for comparison.

The data type \textit{Element} is defined for Cray pointers and contains two integer components \textit{fac\_nr} and \textit{next}. Declaration of the Cray pointers is made and a loop is used to complete the list. Memory is allocated dynamically inside the loop. The command \textit{malloc} reserves memory and assigns an integer address to \textit{tmppek}. The physical addresses are stored as integers in list. The link to the next cell is now stored as an address in the integer \textit{next}. If \( i = 1 \), the first address is stored in \textit{firstpek} and \textit{list}. If \( i > 1 \), the address is stored in \textit{list} and as a link in \textit{tri\%next}. For the Cray pointer list the termination condition lies in setting \textit{tri\%next} = 0 when leaving the loop. This corresponds to a null address or a nullification.
Table 1: Creating a linked list

<table>
<thead>
<tr>
<th>Fortran 90</th>
<th>Cray</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Structures</strong></td>
<td></td>
</tr>
<tr>
<td>Type tElem</td>
<td>Type Element</td>
</tr>
<tr>
<td>Sequence</td>
<td>Sequence</td>
</tr>
<tr>
<td>Integer facnr</td>
<td>Integer fac_nr</td>
</tr>
<tr>
<td>Type(tElem), Pointer :: next</td>
<td>Integer next</td>
</tr>
<tr>
<td>End Type tElem</td>
<td>End Type Element</td>
</tr>
<tr>
<td>contains</td>
<td></td>
</tr>
<tr>
<td>Integer NUMB_FACS,i,ALLOCStat</td>
<td>Integer i</td>
</tr>
<tr>
<td><strong>Pointers declaration</strong></td>
<td></td>
</tr>
<tr>
<td>Type(tElem), Pointer :: firstpek1,list1</td>
<td>Type (Element) first,tmp,tri</td>
</tr>
<tr>
<td>ALLOCATE(firstpek1, STAT=ALLOCStat)</td>
<td>Pointer (firstpek,first), (tmppek,tmp), (list,tri)</td>
</tr>
<tr>
<td>firstpek1%facnr = 1</td>
<td></td>
</tr>
<tr>
<td>list1 =&gt; firstpek1</td>
<td></td>
</tr>
<tr>
<td><strong>Creating linked list</strong></td>
<td></td>
</tr>
<tr>
<td>Do i=2,NUMB_FACS</td>
<td>Do i=1,NUMB_FACS</td>
</tr>
<tr>
<td>ALLOCATE(list1%next)</td>
<td>tmppek= malloc(8)</td>
</tr>
<tr>
<td>list1%next%facnr = i</td>
<td>if(i .EQ. 1)then</td>
</tr>
<tr>
<td>list1 =&gt; list1%next</td>
<td>list= tmppek</td>
</tr>
<tr>
<td></td>
<td>firstpek= list</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>tri%next= tmppek</td>
</tr>
<tr>
<td></td>
<td>list= tri%next</td>
</tr>
<tr>
<td></td>
<td>endif</td>
</tr>
<tr>
<td></td>
<td>tri%fac_nr = 1</td>
</tr>
<tr>
<td><strong>Termination of the list</strong></td>
<td></td>
</tr>
<tr>
<td>End Do</td>
<td>tri%next = 0</td>
</tr>
<tr>
<td>NULLIFY(list1%next)</td>
<td>End Do</td>
</tr>
</tbody>
</table>
8 Conclusion

One way to generate CAD objects is to use regular Fortran code that produces the CAD format without any graphical interface. These objects can be displayed on a pixel screen by means of ray tracing. Ray tracing is computationally intensive. In order to speed up the calculations, spatial sub-division, in the form of either a Grid or BSP was used.

Data structures with Fortran 90 pointers are used in the implementation. Modules are used in such a way that pointers in the parameter lists of subroutines are contained within the module. Conversion from Cray pointers to Fortran 90 pointers is carried out and the code works as intended. This type of conversion improves the portability of the code since f90 pointers are a part of the standard. Conversion from Cray to f90 pointers can be made in steps where the two implementation are used in parallel as a means for debugging.
References


http://www.amath.unc.edu/sysadmin/DOC4.0/fortran/user_guide/C_f90.doc.html


http://www.ibiblio.org/pub/languages/fortran/ch2-16.html


Appendix A The program Polgen

program polgen
  c
  c This code generates a polygon file and a dxf file
  c
  real*8 xyz(2,3),hj(5,3),th(2),fi(2),Pi,r(3), r_func
  real*8 rad, the, phi
  integer num, j, i, laynum, ant, n, ith, ifi, i2, j2, nfi, laynum_1
  character q(0:15)
  character*3 w3(10000), w
  logical TH_CRIT, FI_CRIT

  Pi = 3.141592653589793 D0
  do j=1,3
    xyz(1,j)= 1. d10
    xyz(2,j)=-1. d10
  enddo
  call gen_hex(w3) ! generate Entity handles

  open(2, file='ut.pol')
  rewind 2
  write(2,*) 'Polygonfil' ! header pol
  call system('cp start_file_R12 ut.dxf ') ! start_file
  open(3, file='ut.dxf', access='APPEND')
  call start_Entities_R12 ! header Entities

  n= 30
  nfi = 2*n
  write(6,*) 'Number of polygons: ', n*nfi
  laynum= 0
  num= 0 ! facet number
  do ith= 0, n-1
    th(1)= Pi*ith/dfloat(n)
    th(2)= Pi*(ith+1)/dfloat(n)
    the= (th(1)+th(2))/2. d0
    do ifi= 0, nfi-1
      num= num+1
      fi(1)= 2*Pi*ifi/dfloat(nfi)
      fi(2)= 2*Pi*(ifi+1)/dfloat(nfi)
      phi= (fi(1)+fi(2))/2. d0
      if((ith.EQ.0 .OR. ith.EQ.n-1) then
        ant= 3
      else
        ant= 4
      endif
      do i=1, ant
        if(i .EQ. 1) then
          i2=1; j2= 1
        elseif(i .EQ. 2) then
          i2=2; j2= 1
        elseif(i .EQ. 3) then
          i2=2; j2= 2
        if((ith .EQ. n-1) then
          i2=1; j2= 2
        endif
        else
          i2=1; j2= 2
        endif
        call r_th_fi(th(i2),fi(j2),r) ! compute corners
do j=1,3
    hj(i,j)= r(j)        ! set corner
enddo
enddo

TH_CRIT = the.LT.20.d0 .OR. the.GT.2.3d0
FI_CRIT = phi.LT.20.d0 .OR. phi.GT.3.0d0
if((TH_CRIT .AND. FI_CRIT) then  ! select part
    laynum_1 = laynum
else
    laynum_1= 1
endif

write(2,*), 3*ant+2  ! write polygon file
write(2,*), laynum_1
write(2,*), num       ! layer number
write(2,*), num

do 40 i= 1, ant
    do 41 j=1,3
        if(hj(i,j) .ge. xyz(2,j)) xyz(2,j)= hj(i,j)  ! max x,y,z
        if(hj(i,j) .lt. xyz(1,j)) xyz(1,j)= hj(i,j)  ! min x,y,z
    continue

    write (2,*), sngl(hj(i,1)), sngl(hj(i,2)), sngl(hj(i,3))

40 continue

if(laynum_1 .EQ. 0) then
    w = w3(50+num)  ! avoid low handles
    call wri3D_h_R12(hj,ant,num,w,laynum_1)  !3DFACE objects
endif
enddo
enddo

write(2,*), 0  ! footer pol
write(2,*), 'COMBI'      ! footer dxf
write(6,*), 'largest Ent. handle = ', w, '< HANDSEED=A01'

if((ith.EQ.n-1 .AND. ifi.EQ.n-1)
    then
    write(6,*), 'Limits: '
    do j=1,3
        write(6,*), (sngl(xyz(i,j)),i=1,2)
    enddo
    write(6,*), 'Default Aimingpoint:',
        &((sngl(xyz(2,j)+xyz(1,j))/2.d0),j=1,3)
    write(6,*), 'Widths Dx, Dy, Dz:',
        &((sngl(abs(xyz(2,j)-xyz(1,j))),j=1,3)
close(2)

20 format(40A)  ! stop

-------------------------------------------------------------------------------------

subroutine start_Entities_R12
write(3,20) 'SECTION'  ! start Entities
write(3,20) '2'
write(3,20) 'ENTITIES'
write(3,20) '0'
20 format(40A)
return
end
-------------------------------------------------------------------------------------
subroutine stop_Entities_R12
write(3,20) 'ENDSEC' ! end of Entities
write(3,20) '0' ! footer Entities
write(3,20) 'EDF' ! end of file
20 format(40A)
return
end

subroutine gen_hex(w3)
c create 3 digit hex string
integer num,i,j,n
character*3 w3(10000)
character q(0:15)
q(0)= '0 '; q(1)= '1 '; q(2)= '2 '; q(3)= '3 '; q(4)= '4 '
qu(5)= '5 '; q(6)= '6 '; q(7)= '7 '; q(8)= '8 '; q(9)= '9 '
qu(10)= 'A ';q(11)= 'B ';q(12)= 'C ';q(13)= 'D ';q(14)= 'E '
& q(15)= 'F'
um= 0
do i=0,15
do j=0,15
do n=0,15
w3(num)= q(i)//q(j)//q(n)
endo
dendo
dendo
return
end

subroutine wri3D_h_R12(hj ,ant ,num ,w, laynum)
c write 3DFACE object; AutoCAD version R12
integer ant,i,j, grk (5,3),num,laynum
real*8 hj (5,3)
character*3 w
grk (1,1)=10; grk (1,2)=20; grk (1,3)=30 !corner 1
grk (2,1)=11; grk (2,2)=21; grk (2,3)=31 !corner 2
grk (3,1)=12; grk (3,2)=22; grk (3,3)=32 !corner 3
grk (4,1)=13; grk (4,2)=23; grk (4,3)=33 !corner 4
write(3,20) '3DFACE' !header 3DFACE
write(3,20) ' 5' !5
write(3,20) w !layer; Group code 8
write(3,24) laynum !layer number; 0
do i=1 , ant
do j=1,3
write(3,23)grk(i,j) !corner code
if(hj(i,j) .GT. 0. d0) then
write(3,21)hj(i,j) !corner
else
write(3,22)hj(i,j)
endif
endo
dendo
write(3,20) '0' !footer 3DFACE
20 format(40A)
21 format(F12.10)
22 format(F12.9)
23 format(13)
24 format(11)
return
end
subroutine r_th_fi(th,fi,r)
  c Specify geometry of object
  real*8 th,fi,r(3),th0,th1,rc,pi,
  ellipse,eps,th2,a,ths,d,d_,thc
  logical SPHERE,SPHEROID,CONESPHERE_CUSP,CONESPHERE
  logical CYLINDER
  external ellipse
  pi= 3.141592653589793d0

  SPHERE=.false.
  SPHEROID=.true.
  CONESPHERE_CUSP=.false.
  CONESPHERE=.false.
  CYLINDER=.false.

  if (SPHERE) then
    th1= th; rc= 1.d0; d= 0.d0
  elseif (SPHEROID) then
    a= 0.5d0; eps = -0.5d0; th2= 0.0d0
    th1= th; rc= ellipse(a,th,eps,th2); d= 0.d0
  elseif (CONESPHERE_CUSP) then
    th0= 0.3d0; d= 0.d0
    if(th<th0) then
      th1= th; rc= (Pi-th)/(Pi-th0)
    else
      th1= th0; rc= (Pi-th)/(Pi-th0)
    endif
  elseif (CONESPHERE) then
    ths= 0.3; th0= ths+ Pi/2.d0
    if(th<th0) then
      th1= th; rc= 1.d0; d= 0.d0
    else
      d = 1.d0/sin(th)
      th1= ths; rc= d*cos(ths)*(Pi-th)/(Pi-th0)
    endif
  elseif (CYLINDER) then
    th0= 1.4; d_= 3.d0; thc= Pi-th0
    if(th<th0) then
      th1= th; rc= 1.d0; d=d_
    elseif (th>thc) then
      th1= th; rc= 1.d0; d=d_
    else
      th1= th0; rc= 1.d0
      d = -d_+ 2*(th-th0)/(thc-th0)*(d_*cos(th0))
    endif
  else
    stop 'no object specified'
  endif

  r(1)= sin(th1)*cos(fi)*rc
  r(2)= sin(th1)*sin(fi)*rc
  r(3)= cos(th1)*rc- d
  return
end
function ellipse(a,th,eps,th0)
c  elliptical boundary in polar form; th0 rotation
real*8 a,l1,l2,eps,th0,Pi,limit
real*8 th,ellipse,th_num,ta,Rx,Ry,x,y,xp,yp,R
parameter(Pi=3.141592653589793D0, limit= 1.d-20)

  l1= a*(1.d0+ eps)
  l2= a*(1.d0- eps)
  th_num= th- th0
  if( abs(th_num).LT. limit .OR.
    $ abs(abs(th_num)-Pi/2.d0).LT. limit .OR.
    $ abs(abs(th_num)-Pi).LT. limit) then
    th_num= th_num +2* limit ! avoid sing. in tan
  endif
  ta= sin(th_num)/cos(th_num)
  Rx= sqrt((l2/l1)**2+ ta **2)
  Ry= sqrt((l1/l2)**2+ 1/ ta **2)
  x= l2/Rx ! polar form
  y= l1/Ry
  R= sqrt(x**2+ y**2)
  ellipse= R
return
end

c--------------------------------------------------------

39
Appendix B  Examples

B.1  Linked list

Program Llist
 Type tElem
   INTEGER :: facnr
   Type( tElem ), Pointer :: next
 End type tElem

INTEGER i, AllocStat
N = 10
Type(tElem), Pointer :: firstpek1, list1

Nullify (firstpek1)
ALLOCATE( firstpek1, STAT=AllocStat)
Nullify (firstpek1%next)
firstpek1%facnr = 1
list1 => firstpek1

Do i=2,N
ALLOCATE (list1%next)
  list1%next%facnr = i
  list1 => list1%next
End Do

Print *, 'List elements are:
list1 => firstpek1

Do while( associated( list1 ) )
  Print *, list1%facnr
  list1 => list1%next
End Do

End Program Llist
B.2 The BSP tree

Program tree
Implicit none
Type Record sequence
REAL*8 bmin(3), bmax(3) ! Variables for the boxes in the tree.
Type (Record), Pointer :: left
INTEGER :: generation
Type (Record), Pointer :: right
End Type Record

Type (Record), Pointer :: root
INTEGER count, level, n
n = 16
level = 0
count = 0
root => generate_tree(n, level)
call tree_print(root, count)
contains

Recursive function generate_tree(n, level) result (t)
imPLICIT none
INTEGER n, level
Type (Record), Pointer :: t
allocate(t)
level = level + 1
!generation = level
If (n <= 2) then
nullify(t%left)
Else
t%left => generate_tree(n/2, level)
End If
If (n <= 2) then
nullify(t%right)
Else
t%right => generate_tree(n/2, level)
End If
End Function

Recursive subroutine tree_print(t, count)
imPLICIT none
Type(Record), Pointer :: t
INTEGER count, i
count = count + 1
If (associated(t)) then
call tree_print(t%right, count)
write(*,*)' ',i=0,count-1),t%generation
call tree_print(t%left, count)
End If
count = count - 1
End subroutine

End program