FlockMC: Monte Carlo Particle Simulator

Gagarine Yaikhom

May 26, 2014

Contents

CHAPTER 1

Introduction

The *Monte Carlo Simulator*, `MCS` in short, is a simulation package for high-energy physics. It uses Monte Carlo techniques for the randomised simulation of particle interactions. The system borrows concepts defined in the Geant4 system; however, they have been reimplemented in ISO C99 using new data structures and algorithms with the aim that the system will be ported eventually to GPGPUs.

The `MCS` system is in effect an event processor. The user specifies the number of events they wish to simulate, and provides the system with the necessary details to process each of the events. These details include procedures and data for generating the particles for each event, and the *physics processes* required by the type of the particle being simulated and the properties of the materials involved in each of the interactions. We are only interested in interactions that are happening inside a closed three-dimensional system, referred to as the *world*. Since the world is defined by the components present, the user must provide the geometry of the components in addition to the properties of the materials the components are made of. In this document, these input data will be referred to *simulation parameters*.

Once the `MCS` system is supplied with a valid set of simulation parameters, it processes the geometries and creates a data structure that allows efficient location of an interaction point. This is an important step because, to process a large number of interactions rapidly, we must efficiently find the location inside the world where the interactions are happening, as it determines the materials required by the processes.

We then define the *particle gun*, which generates the required number of *primary particles* for each of the events. Primary particles are particles that are generated explicitly by a particle gun, and *secondary particles* are those that are generated as a result of particle-matter interaction. Since multiple primary particles may be generated by an event, a *vertex* is defined for each event. For any given event the associated vertex provides the location of the particle gun, so that all of the particles generated originates from this location. When the particle gun is activated, a set of primary particles are generated. These particles have the same origin, as defined by the vertex location, however, they carry different properties, such as energy, momentum, etc. These values are set using a *random number generator*.

Once a particle has been generated, it travels through the materials until it either comes to a stop, or exits the closed three-dimensional world. Throughout this journey, the *trajectory* of the particle changes depending on its interaction with the materials. Hence, the `MCS` system uses a *tracker* to track each of the particles throughout its journey. A snapshot of the particle in its trajectory is referred to as a *track*, and is processed independently of any previous tracks. In other words, a track does not remember its past. Hence, to keep a record of the trajectory, the tracker records old tracks as new ones are derived. Each of these points on the trajectory is referred to as the *step point*, and the process of moving to the next step point as *stepping*. Hence, we invoke multiple stepping commands to chart the entire trajectory of a particle.

To move a particle from one track to the next, each stepping decides the length with which the particle must progress. This is referred to as a *step*. Each step has a start point and an end point. These points store information for retrieving the material properties that are required by each of the physics processes that are valid in that stepping. The length of a step is determined by the process which requires the shortest space-time *interaction length*. In other words, the next track must be located in space-time so that all of the valid physics processes are applicable.

We use a *stack* to store all of the unprocessed particles, both primary and secondary. At the beginning of each event, each of the primary particles generated by the particle gun are first converted to a track, so that the tracker can process them. Then, the tracks are all pushed into the *track stack*. This stack is then passed to the tracker, which continuously pops a track from the track stack and charts the particle's trajectory. If secondary particles are generated as a result of an interaction, these are first converted to a track, and then pushed into the track stack. In other words, if we consider the entire simulation as a tree, where the root of the tree represents the particle gun, the nodes represent the particle-matter interactions, and the leaves represent interactions which resulted in the particle to either stop, or exit the closed three-dimensional world, then the tracker is simply undergoing a *depth-first tree traversal*.

CHAPTER 2

History

The *Monte Carlo Simulator* project began in June 2011, when Dr. Gagarine Yaikhom was a WIMCS Research Fellow under Prof. David W. Walker at Cardiff University. This project was conceived as a study on the parallelisation of Monte Carlo particle simulations that would find eventual use in radiotherapy treatment planning. The initial aim of the project was to port the Geant4 system to run on Nvidia Tesla GPUs. However, after studying the Geant4 code-base, by the end of July 2011 it became clear that it will be prohibitive to port the entire Geant4 system given the short duration of the funding left (6 months, from August 2011 until January 2012) and that only Dr. Yaikhom will be carrying out the design and implementation. We therefore decided to use the Geant4 system only as a guideline system architecture, and to reimplement the performance-intensive concepts and their dependencies for a simplified simulator. This implementation, which began on 5 August 2011, uses data structures and algorithms that are designed for multithreaded parallelisation on multicore GPUs and CPUs.

Forward-declare types of several data structures, so that we can easily reshuffle the sections for easy exposition, while avoiding compiler error.

3 ⟨ Forward declare functions 3 ⟩ ≡

```
void process_and_register_solid(CSG_Node *root);
Containment recursively_test_containment(CSG_Node *root, Vector v);
```

This code is used in chunk 361.

CHAPTER 3

Error messaging

This section defines data-structures and functions for handling and reporting errors. There are three message categories, which are printed using the following macros:

- We use the *fatal* macro to advise the user of irrecoverable errors. They are usally commnicated before the application is about to exit due to the errors. Fatal messages must provide enough information to help the user diagnose the issues that caused the exit. Messages communicated with the *fatal* macro should never is suppressed, even when *verbose* $\equiv$ *true*.
- We use the *warn* macro to advise the user of non-fatal recoverable issues, such as wrong input data. They are usually communicated without exiting the application. Messages communicated with the *warn* macro are suppressed when *verbose* $\equiv$ *false*.
- We use the *info* macro to advise the user on the general state of the application, for instance, the current stage in the processing. They are usually communicated without exiting the application. Messages communicated with the *info* macro are suppressed when *verbose* $\equiv$ *false*.

NOTE: Some of the messaging facilities are defined as *variadic macros*, using the ISO C99 syntax. Please ensure that your compiler fully supports variadic macros; otherwise, redefine these macros as functions.

4 ⟨ Global variables 4 ⟩ ≡

```
#define fatal(...) fprintf (stderr, __VA_ARGS__)
#define warn(...) if (verbose) fprintf(stderr, __VA_ARGS__)
#define info(...) if (verbose) fprintf(stderr, __VA_ARGS__)
```

```
bool verbose = false;   /* verbose output if true */
```

See also chunks 25, 41, 55, 67, 93, 97, 156, 191, 308, 347, and 368.

This code is used in chunk 361.

Part I

Common

This section defines common types, physical constants, data-structures and functions that are used in various sections.

We specify Boolean variables using the **bool** data type.

6 ⟨ Type definitions 6 ⟩ ≡
```
typedef enum {
   false = 0, true
} bool;
```
See also chunks 11, 24, 32, 43, 44, 57, 58, 59, 69, 196, 231, 266, 267, 269, 287, 301, 309, 314, 320, 328, 335, 338, 350, and 351.

This code is used in chunk 361.

The following are mathematical constants that are use throughout.

7
```
#define PI 3.14159265358979323846
#define TWICE_PI 6.283185307
#define RADIAN_TO_DEGREE (180.0/PI)     /* RADIAN_TO_DEGREE = 180/π */
#define DEGREE_TO_RADIAN (PI/180.0)
```

Function *convert_radian_to_degree*(*angle*) determines the positive angle in degrees that is equivalent to the specified *angle* in radians.

8 ⟨ Global functions 8 ⟩ ≡
```
double convert_radian_to_degree(double angle)
{
   angle *= RADIAN_TO_DEGREE;
   if (angle < 0.0) angle += 360.0;     /* positive angle required */
   return angle;
}
```
See also chunks 9, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 26, 27, 28, 29, 30, 33, 35, 40, 56, 68, 70, 71, 72, 73, 74, 76, 159, 161, 166, 167, 168, 180, 184, 192, 193, 199, 200, 203, 208, 209, 218, 219, 233, 234, 237, 238, 239, 242, 243, 246, 250, 251, 258, 272, 278, 279, 284, 285, 288, 289, 290, 292, 293, 300, 302, 303, 305, 307, 310, 311, 312, 315, 316, 317, 318, 319, 321, 322, 323, 324, 325, 333, 336, 339, 342, 348, 349, 352, 353, 357, and 358.

This code is used in chunk 361.

Function *roundup_pow2*(*v*) rounds up a positive number to the nearest power of two.

9 ⟨ Global functions 8 ⟩ +≡
```
unsigned roundup_pow2(unsigned v)
{
   if (v ≡ 0) return 1;
   if (v & (v − 1)) {     /* check if already power of two */
      unsigned k;
      ⟨Find non-zero most significant bit 10⟩;
      return $(1 \ll (k+1))$;
    }
    return $v$;
  }
```

10 ⟨Find non-zero most significant bit 10⟩ ≡

```
  for $(k = \textbf{sizeof}(\textbf{unsigned}) * 8 - 1;\ ((1 \ll k)\ \&\ v) \equiv 0;\ {-}{-}k)$ ;
```

This code is used in chunk 9.

3.1 Vector

This section defines data-structures and functions for representing and manipulating three-dimensional vectors.

In MCS, we use the *homogeneous coordinate system* introduced by August Ferdinand Möbius in his work *Der barycentrische Calcül* [Leipzig, **1827**]. By expressing vectors using homogeneous coordinates, we simplify *affine transformations* of a vector, such as translation, rotation and scaling, as these can be expressed efficiently as a left-hand multiplication of the vector by a 4×4 affine transformation matrix.

The homogeneous coordinates of a finite point (x', y', z') in the three-dimensional cartesian volume are any four numbers (x, y, z, w) for which $x' = x/w$, $y' = y/w$, and $z' = z/w$. Before applying any affine transformation matrix to a vector, we must ensure that $w = 1$. We shall refer to this as the w-component.

11 ⟨ Type definitions 6 ⟩ +≡
```
typedef double Vector[4];
```

Function *vector_print*(f, v) prints the components of vector v to the I/O stream pointed to by f. Vector v is left unmodified.

12 ⟨ Global functions 8 ⟩ +≡
```
void vector_print(FILE *f, const Vector v)
{
  fprintf(f, "(%lf,␣%lf,␣%lf,␣%lf)", v[0], v[1], v[2], v[3]);
}
```

Function *vector_zero*(v) modifies the vector v by setting all of its components to zero, except for its w-component, which is set to 1.

13 **#define** ZERO_VECTOR {0.0, 0.0, 0.0, 1.0}

⟨ Global functions 8 ⟩ +≡
```
void vector_zero(Vector v)
{
  v[0] = v[1] = v[2] = 0.0;
  v[3] = 1.0;
}
```

Function *vector_homogenise*(v) modifies the vector v, so that its w-component equals 1.

```
14 ⟨ Global functions 8 ⟩ +≡
   void vector_homogenise(Vector v)
   {
     if (1.0 ≡ v[3]) return;     /* already homogenised */
     if (0.0 ≠ v[3]) {
       v[0] /= v[3];
       v[1] /= v[3];
       v[2] /= v[3];
       v[3] = 1.0;
     }
   }
```

Function *vector_copy_f*(u, v) modifies vector u by copying all of the components of vector v to vector u. Vector v is left unmodified. We may prefer to use the macro *vector_copy*(X, Y) instead.

```
15 #define vector_copy(X, Y)   memcpy((X), (Y), 4 * sizeof(double))
   ⟨ Global functions 8 ⟩ +≡
   void vector_copy_f(Vector u, const Vector v)
   {
     u[0] = v[0];
     u[1] = v[1];
     u[2] = v[2];
     u[3] = v[3];
   }
```

Function *vector_magnitude*(v) returns the *vector magnitude* of the vector v. Vector v is left unmodified.

```
16 ⟨ Global functions 8 ⟩ +≡
   double vector_magnitude(const Vector v)
   {
     return sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
   }
```

Function *vector_normalize*(v, r) *normalises* the vector v and stores the result in r. Vector v is left unmodified. The normalised vector r is an *unit vector* with magnitude 1. This function assumes that the w-component of v is already 1.

```
17 ⟨ Global functions 8 ⟩ +≡
   void vector_normalize(const Vector v, Vector r)
   {
     double m = vector_magnitude(v);     /* assumes r[3] = 1.0 */
```

```
    if (m > 0.0) {
       r[0] = v[0]/m;
       r[1] = v[1]/m;
       r[2] = v[2]/m;
    }
    else {
       r[0] = 0.0;
       r[1] = 0.0;
       r[2] = 0.0;
    }
 }
```

Function *vector_difference*(u, v, r) calculates the *vector difference* by subtracting vector v from vector u, and stores the result in vector r. Both vectors u and v are left unmodified. This function assumes that the w-components in v and u are already 1.

18 ⟨ Global functions 8 ⟩ +≡

```
 void vector_difference(const Vector u, const Vector v, Vector r)
 {
    r[0] = u[0] − v[0];
    r[1] = u[1] − v[1];
    r[2] = u[2] − v[2];
    r[3] = 1.0;
 }
```

Function *vector_dot*(u, v) returns the *dot* (also known as the *scalar*) product of the two vectors u and v. Vectors u and v are left unmodified. Vector dot products are *commutative*—the order of the parameters are irrelevant.

19 ⟨ Global functions 8 ⟩ +≡

```
 double vector_dot(const Vector u, const Vector v)
 {
    return u[0] ∗ v[0] + u[1] ∗ v[1] + u[2] ∗ v[2];
 }
```

Function *vector_cross*(u, v, r) calculates the *cross* product between vector u and vector v, and stores the result in vector r. Vectors u and v are left unmodified.

Vector r is perpendicular to both v and u, with a direction given by the *right-hand rule* (when u and v respectively point towards the index finger and the middle finger, r points in the direction of the thumb). The *magnitude* of r is equal to the area of the parallelogram that the vectors span from u towards

v. Vector cross products are *noncommutative*—the order of the parameters are significant.

20 ⟨ Global functions 8 ⟩ +≡
```
void vector_cross(const Vector u, const Vector v, Vector r)
{
   r[0] = (u[1] * v[2] − u[2] * v[1]);
   r[1] = (u[2] * v[0] − u[0] * v[2]);
   r[2] = (u[0] * v[1] − u[1] * v[0]);
}
```

Function *vector_angle_radian*(u, v) returns the angle θ in radians between the vectors u and v, where rotation started at vector u. The value of θ is in the range $[0, 2\pi]$. Vectors u and v are left unmodified.

21 #define TWICE_PI 6.283185307

⟨ Global functions 8 ⟩ +≡
```
double vector_angle_radian(const Vector u, const Vector v)
{
   Vector a, b, c = ZERO_VECTOR;
   double angle;
   vector_normalize(u, a);
   vector_normalize(v, b);
   angle = acos(vector_dot(a, b));
   vector_cross(u, v, c);
   if (c[3] < 0.0) return (TWICE_PI − angle);
   return angle;
}
```

Function *vector_angle_degree*(u, v) returns the angle θ in degrees between the vectors u and v, where rotation began at vector u. The value of θ is in the range $[0°, 360°]$. Vectors u and v are left unmodified.

22 ⟨ Global functions 8 ⟩ +≡
```
double vector_angle_degree(const Vector u, const Vector v)
{
   return RADIAN_TO_DEGREE * vector_angle_radian(u, v);
}
```

Function *vector_distance*(u, v) returns the distance between the three-dimensional points represented by the vectors u and v. Vectors u and v are left unmodified.

23 ⟨ Global functions 8 ⟩ +≡

```
double vector_distance(const Vector u, const Vector v)
{
  double x, y, z;
  x = u[0] − v[0];
  y = u[1] − v[1];
  z = u[2] − v[2];
  return sqrt(x ∗ x + y ∗ y + z ∗ z);
}
```

3.2 Matrix

This section defines data-structures and functions for representing and manipulating matrices.

In MCS, we use a 4×4 matrix to represent affine transformations such as translation, rotation and scaling. These matrices are applied to vectors in homogeneous coordinates, with w-component equal to 1, to determine the transformed resultant vector.

24
```
⟨ Type definitions 6 ⟩ +≡
  typedef double Matrix[4][4];     /* a 4 × 4 matrix */
```

The *identity matrix* represents a null transformation. Applying an affine transformation represented by an identify matrix leaves the original vector unmodified. All new affine transformation matrices must be initialised to the IDENTITY_MATRIX. When we wish to convert an existing matrix into the identity matrix, we simply copy the values stored in the immutable global variable *identity_matrix*.

25
```
#define IDENTITY_MATRIX
        {{1.0, 0.0, 0.0, 0.0}, {0.0, 1.0, 0.0, 0.0}, {0.0, 0.0, 1.0, 0.0}, {0.0, 0.0, 0.0, 1.0}}
⟨ Global variables 4 ⟩ +≡
  const Matrix identity_matrix = IDENTITY_MATRIX;
```

Function *matrix_print*(f, m, nr, nc, t) prints the $nr \times nc$ matrix m to the I/O stream pointed to by f using an indentation of t blank tabs.

26
```
⟨ Global functions 8 ⟩ +≡
  void matrix_print(FILE *f, Matrix m, uint8_t nr, uint8_t nc, uint8_t t)
  {
    uint8_t i, j, k;
    for (i = 0; i < nr; ++i) {
      for (k = 0; k < t; ++k) fprintf(f, "\t");
      fprintf(f, "|␣");
      for (j = 0; j < nc; ++j) fprintf(f, "%8.3lf␣", m[i][j]);
      fprintf(f, "|\n");
    }
  }
```

Function *matrix_copy_f*(u, v) modifies the matrix u by copying all of the 16 elements from matrix v to matrix u. Matrix v is left unmodified.

27
```
#define matrix_copy(X, Y)   memcpy((X), (Y), 16 * sizeof(double))
```

```
⟨ Global functions 8 ⟩ +≡
  void matrix_copy_f (Matrix u, Matrix v)
  {
    u[0][0] = v[0][0];
    u[0][1] = v[0][1];
    u[0][2] = v[0][2];
    u[0][3] = v[0][3];
    u[1][0] = v[1][0];
    u[1][1] = v[1][1];
    u[1][2] = v[1][2];
    u[1][3] = v[1][3];
    u[2][0] = v[2][0];
    u[2][1] = v[2][1];
    u[2][2] = v[2][2];
    u[2][3] = v[2][3];
    u[3][0] = v[3][0];
    u[3][1] = v[3][1];
    u[3][2] = v[3][2];
    u[3][3] = v[3][3];
  }
```

Function *matrix_multiply* (a, b, c) multiplies the 4×4 matrix a to the 4×4 matrix b, and stores the result in matrix c. Matrix multiplication is *noncommutative*—the order of the parameters are significant.

```
28 ⟨ Global functions 8 ⟩ +≡
  void matrix_multiply (Matrix a, Matrix b, Matrix c)
  {
    c[0][0] = a[0][0] * b[0][0] + a[0][1] * b[1][0] + a[0][2] * b[2][0] + a[0][3] * b[3][0];
    c[0][1] = a[0][0] * b[0][1] + a[0][1] * b[1][1] + a[0][2] * b[2][1] + a[0][3] * b[3][1];
    c[0][2] = a[0][0] * b[0][2] + a[0][1] * b[1][2] + a[0][2] * b[2][2] + a[0][3] * b[3][2];
    c[0][3] = a[0][0] * b[0][3] + a[0][1] * b[1][3] + a[0][2] * b[2][3] + a[0][3] * b[3][3];
    c[1][0] = a[1][0] * b[0][0] + a[1][1] * b[1][0] + a[1][2] * b[2][0] + a[1][3] * b[3][0];
    c[1][1] = a[1][0] * b[0][1] + a[1][1] * b[1][1] + a[1][2] * b[2][1] + a[1][3] * b[3][1];
    c[1][2] = a[1][0] * b[0][2] + a[1][1] * b[1][2] + a[1][2] * b[2][2] + a[1][3] * b[3][2];
    c[1][3] = a[1][0] * b[0][3] + a[1][1] * b[1][3] + a[1][2] * b[2][3] + a[1][3] * b[3][3];
    c[2][0] = a[2][0] * b[0][0] + a[2][1] * b[1][0] + a[2][2] * b[2][0] + a[2][3] * b[3][0];
    c[2][1] = a[2][0] * b[0][1] + a[2][1] * b[1][1] + a[2][2] * b[2][1] + a[2][3] * b[3][1];
    c[2][2] = a[2][0] * b[0][2] + a[2][1] * b[1][2] + a[2][2] * b[2][2] + a[2][3] * b[3][2];
    c[2][3] = a[2][0] * b[0][3] + a[2][1] * b[1][3] + a[2][2] * b[2][3] + a[2][3] * b[3][3];
    c[3][0] = a[3][0] * b[0][0] + a[3][1] * b[1][0] + a[3][2] * b[2][0] + a[3][3] * b[3][0];
    c[3][1] = a[3][0] * b[0][1] + a[3][1] * b[1][1] + a[3][2] * b[2][1] + a[3][3] * b[3][1];
    c[3][2] = a[3][0] * b[0][2] + a[3][1] * b[1][2] + a[3][2] * b[2][2] + a[3][3] * b[3][2];
    $c[3][3] = a[3][0] * b[0][3] + a[3][1] * b[1][3] + a[3][2] * b[2][3] + a[3][3] * b[3][3]$;
  }
```

Function *matrix_determinant*(m) returns the *determinant* of a 4×4 matrix. Matrix m is left unmodified.

29 ⟨ Global functions 8 ⟩ +≡

```
  **double** *matrix_determinant*(**Matrix** $m$)
  {
    **return** $m[0][3]*m[1][2]*m[2][1]*m[3][0] - m[0][2]*m[1][3]*m[2][1]*m[3][0] -$
        $m[0][3] * m[1][1] * m[2][2] * m[3][0] + m[0][1] * m[1][3] * m[2][2] * m[3][0] +$
        $m[0][2] * m[1][1] * m[2][3] * m[3][0] - m[0][1] * m[1][2] * m[2][3] * m[3][0] -$
        $m[0][3] * m[1][2] * m[2][0] * m[3][1] + m[0][2] * m[1][3] * m[2][0] * m[3][1] +$
        $m[0][3] * m[1][0] * m[2][2] * m[3][1] - m[0][0] * m[1][3] * m[2][2] * m[3][1] -$
        $m[0][2] * m[1][0] * m[2][3] * m[3][1] + m[0][0] * m[1][2] * m[2][3] * m[3][1] +$
        $m[0][3] * m[1][1] * m[2][0] * m[3][2] - m[0][1] * m[1][3] * m[2][0] * m[3][2] -$
        $m[0][3] * m[1][0] * m[2][1] * m[3][2] + m[0][0] * m[1][3] * m[2][1] * m[3][2] +$
        $m[0][1] * m[1][0] * m[2][3] * m[3][2] - m[0][0] * m[1][1] * m[2][3] * m[3][2] -$
        $m[0][2] * m[1][1] * m[2][0] * m[3][3] + m[0][1] * m[1][2] * m[2][0] * m[3][3] +$
        $m[0][2] * m[1][0] * m[2][1] * m[3][3] - m[0][0] * m[1][2] * m[2][1] * m[3][3] -$
        $m[0][1] * m[1][0] * m[2][2] * m[3][3] + m[0][0] * m[1][1] * m[2][2] * m[3][3]$;
  }
```

Function *matrix_inverse*(m, i) calculates the *inverse* of a 4×4 matrix and stores the result in the 4×4 matrix i. Matrix m is left unmodified.

30 ⟨ Global functions 8 ⟩ +≡

```
  **void** *matrix_inverse*(**Matrix** $m$, **Matrix** $i$)
  {
    **double** $det = $ *matrix_determinant*($m$);
    $i[0][0] = (m[1][2] * m[2][3] * m[3][1] - m[1][3] * m[2][2] * m[3][1] + m[1][3] *$
        $m[2][1] * m[3][2] - m[1][1] * m[2][3] * m[3][2] - m[1][2] * m[2][1] * m[3][3] +$
        $m[1][1] * m[2][2] * m[3][3]) / det$;
    $i[0][1] = (m[0][3] * m[2][2] * m[3][1] - m[0][2] * m[2][3] * m[3][1] - m[0][3] *$
        $m[2][1] * m[3][2] + m[0][1] * m[2][3] * m[3][2] + m[0][2] * m[2][1] * m[3][3] -$
        $m[0][1] * m[2][2] * m[3][3]) / det$;
    $i[0][2] = (m[0][2] * m[1][3] * m[3][1] - m[0][3] * m[1][2] * m[3][1] + m[0][3] *$
        $m[1][1] * m[3][2] - m[0][1] * m[1][3] * m[3][2] - m[0][2] * m[1][1] * m[3][3] +$
        $m[0][1] * m[1][2] * m[3][3]) / det$;
    $i[0][3] = (m[0][3] * m[1][2] * m[2][1] - m[0][2] * m[1][3] * m[2][1] - m[0][3] *$
        $m[1][1] * m[2][2] + m[0][1] * m[1][3] * m[2][2] + m[0][2] * m[1][1] * m[2][3] -$
        $m[0][1] * m[1][2] * m[2][3]) / det$;
    $i[1][0] = (m[1][3] * m[2][2] * m[3][0] - m[1][2] * m[2][3] * m[3][0] - m[1][3] * m[2][0] * m[3][2] + m[1][0] * m[2][3] * m[3][2] + m[1][2] * m[2][0] * m[3][3] - m[1][0] * m[2][2] * m[3][3])/det;$
    $i[1][1] = (m[0][2] * m[2][3] * m[3][0] - m[0][3] * m[2][2] * m[3][0] + m[0][3] * m[2][0] * m[3][2] - m[0][0] * m[2][3] * m[3][2] - m[0][2] * m[2][0] * m[3][3] + m[0][0] * m[2][2] * m[3][3])/det;$
    $i[1][2] = (m[0][3] * m[1][2] * m[3][0] - m[0][2] * m[1][3] * m[3][0] - m[0][3] * m[1][0] * m[3][2] + m[0][0] * m[1][3] * m[3][2] + m[0][2] * m[1][0] * m[3][3] - m[0][0] * m[1][2] * m[3][3])/det;$
    $i[1][3] = (m[0][2] * m[1][3] * m[2][0] - m[0][3] * m[1][2] * m[2][0] + m[0][3] * m[1][0] * m[2][2] - m[0][0] * m[1][3] * m[2][2] - m[0][2] * m[1][0] * m[2][3] + m[0][0] * m[1][2] * m[2][3])/det;$
    $i[2][0] = (m[1][1] * m[2][3] * m[3][0] - m[1][3] * m[2][1] * m[3][0] + m[1][3] * m[2][0] * m[3][1] - m[1][0] * m[2][3] * m[3][1] - m[1][1] * m[2][0] * m[3][3] + m[1][0] * m[2][1] * m[3][3])/det;$
    $i[2][1] = (m[0][3] * m[2][1] * m[3][0] - m[0][1] * m[2][3] * m[3][0] - m[0][3] * m[2][0] * m[3][1] + m[0][0] * m[2][3] * m[3][1] + m[0][1] * m[2][0] * m[3][3] - m[0][0] * m[2][1] * m[3][3])/det;$
    $i[2][2] = (m[0][1] * m[1][3] * m[3][0] - m[0][3] * m[1][1] * m[3][0] + m[0][3] * m[1][0] * m[3][1] - m[0][0] * m[1][3] * m[3][1] - m[0][1] * m[1][0] * m[3][3] + m[0][0] * m[1][1] * m[3][3])/det;$
    $i[2][3] = (m[0][3] * m[1][1] * m[2][0] - m[0][1] * m[1][3] * m[2][0] - m[0][3] * m[1][0] * m[2][1] + m[0][0] * m[1][3] * m[2][1] + m[0][1] * m[1][0] * m[2][3] - m[0][0] * m[1][1] * m[2][3])/det;$
    $i[3][0] = (m[1][2] * m[2][1] * m[3][0] - m[1][1] * m[2][2] * m[3][0] - m[1][2] * m[2][0] * m[3][1] + m[1][0] * m[2][2] * m[3][1] + m[1][1] * m[2][0] * m[3][2] - m[1][0] * m[2][1] * m[3][2])/det;$
    $i[3][1] = (m[0][1] * m[2][2] * m[3][0] - m[0][2] * m[2][1] * m[3][0] + m[0][2] * m[2][0] * m[3][1] - m[0][0] * m[2][2] * m[3][1] - m[0][1] * m[2][0] * m[3][2] + m[0][0] * m[2][1] * m[3][2])/det;$
    $i[3][2] = (m[0][2] * m[1][1] * m[3][0] - m[0][1] * m[1][2] * m[3][0] - m[0][2] * m[1][0] * m[3][1] + m[0][0] * m[1][2] * m[3][1] + m[0][1] * m[1][0] * m[3][2] - m[0][0] * m[1][1] * m[3][2])/det;$
    $i[3][3] = (m[0][1] * m[1][2] * m[2][0] - m[0][2] * m[1][1] * m[2][0] + m[0][2] * m[1][0] * m[2][1] - m[0][0] * m[1][2] * m[2][1] - m[0][1] * m[1][0] * m[2][2] + m[0][0] * m[1][1] * m[2][2])/det;$
}
```

3.3 Memory management

Memory space must be allocated to store the data that are required by the simulations. This include data structures for representing the simulation world, the particles and their trajectories, the physics tables etc. To reduce the number of system calls required to allocate memory, and to simplify deallocation of memory during clean-up, we borrow the memory management scheme described by Donald E. Knuth in the book *The Stanford GraphBase* [Addison-Wesley (**1993**)].

With the concept of memory areas, we separate the conceptual representation of the data from their actual storage in physical memory. We store all of the raw data inside a linked list of memory blocks, each block storing a collection of data of the same data type. Within each data item, we store the information that are required to process the higher-level conceptual representations.

Each memory area is represented by a variable of type *Area*. Before using a memory area s, we must first initialise it by using the C/ macro *mem_init*(s).

32
```
#define mem_init(s)  *s = Λ
⟨ Type definitions 6 ⟩ +≡
  typedef struct memory_area {
    char *first;     /* pointer to the first usable location */
    struct memory_area *next;
        /* pointer to previously allocated block */
  } *Area[1];
```

Function *mem_free*(s) frees the memory blocks currently allocated under the memory area s. We start at the head of the linked list and free each of the blocks. At the end of this function, the memory area variable s will be in a state similar to the one when it was first initialised.

33
```
⟨ Global functions 8 ⟩ +≡
  void mem_free(Area s)
  {
    Area t;
    while (*s) {
      *t = (*s)→next;
      free((*s)→first);
      *s = *t;
    }
  }
```

To allocate storage space to fit n elements of a given data type t under the memory area s, we use the C/ macro *mem_typed_alloc*(n, t, s).

34 **#define** *mem_typed_alloc*(n, t, s) $(t *)$ *mem_alloc*$((n) *$ **sizeof** $(t), s)$

Function *mem_alloc*(n, s) allocates a block of storage space under the memory area s that will fit n consecutive bytes of data. In addition to the raw data, a block must also allocate enough space to store the two pointers defined by a memory area. These are used to implement the linked list of memory blocks. The memory area pointers are stored at the end of each block. To improve efficiency of memory allocation, we allocate a block as a consecutive array of 256 bytes.

```
35 ⟨ Global functions 8 ⟩ +≡
  char *mem_alloc(size_t n, Area s)
  {
    size_t m = sizeof(char *);      /* size of a pointer variable */
    char *block;      /* address of the new block */
    Area t;
    ⟨ Check if the requested size of the block is valid 36 ⟩;
    ⟨ Allocate space as a consecutive array of 256 bytes each 37 ⟩;
    ⟨ Add new block to the area by updating the linked list 38 ⟩;
    return block;
  }
```

For a request to be valid, the number of bytes $n > 0$ and $n \leq (2^{16} - 1) - 2m$.

```
36 ⟨ Check if the requested size of the block is valid 36 ⟩ ≡
  if (1 > n ∨ (#ffff00 − 2 * m) < n) {
    return Λ;
  }
```

This code is used in chunk 35.

We first round up, if necessary, the number of bytes n so that it is a multiple of the size of a pointer variable. This ensures that the area pointers are placed at a location n bytes after the pointer returned by *calloc*( ). No matter what that address is, the placement after n bytes is unaffected as long as the size of a pointer variable is a divisor of 256, since we allocate the space as an array of 256 bytes each.

```
37 #define round_up_mult(x, y)   ((((x) + (y) − 1)/(y)) * (y))
⟨ Allocate space as a consecutive array of 256 bytes each 37 ⟩ ≡
  n = round_up_mult(n, m);      /* round up n to a multiple of m */
  block = (char *) calloc((unsigned)((n + 2 * m + 255)/256), 256);
```

This code is used in chunk 35.

To update the linked list, we first locate the address of the area pointers. Then, we append the new block at the beginning of the linked list that is currently pointed by the memory area s.

38 ⟨ Add new block to the area by updating the linked list 38 ⟩ ≡
```
if (block) {
   *t = (struct memory_area *)(block + n);
   (*t)->first = block;
   (*t)->next = *s;
   *s = *t;
}
```
This code is used in chunk 35.

To allocate a two-dimensional array which will fit $r \times c$ elements of a given data type t under the memory area s, we use the C/ macro *mem_typed_alloc2d*(r, c, t, s).

39 **#define** *mem_typed_alloc2d*(r, c, t, s)
```
        ( t ** ) mem_alloc_2d((r), (c), sizeof (t), s)
```

40 ⟨ Global functions 8 ⟩ +≡
```
char **mem_alloc_2d(uint32_t r, uint32_t c, size_t s, Area a)
{
   char **p = Λ;
   uint32_t i;
   p = (char **) mem_alloc(r * sizeof(char *), a);
   if (Λ ≡ p) return Λ;
   for (i = 0; i < r; ++i) {
      p[i] = mem_alloc(c * s, a);
      if (Λ ≡ p[i]) return Λ;
   }
   return p;
}
```

We shall use only one memory area for the entire application.

41 ⟨ Global variables 4 ⟩ +≡
```
Area mem_phase_one = {Λ};      /* initialisation */
Area mem_phase_two = {Λ};      /* persistent */
```

Part II

Constructive Solid Geometry

During a simulation event, the tracking of a particle's trajectory happens inside a closed three-dimensional simulation world composed of solids. Each of these solids is defined by a *binary tree* data-structure, which is built recursively bottom-up from primitive convex solids through affine transformations and boolean combination of existing solids. A simulation world may be composed of several non-intersecting solids, and hence, it is best represented by a *forest* of binary trees. Once the forest is ready, we pre-process it to generate a compact data-structure that is optimised for efficient processing during particle tracking.

We use *Constructive Solid Geometry* to represent solids. Most of the concepts used in this implementation are based on the books *Geometric and Solid Modelling: An Introduction* by Christoph M. Hoffman [Morgan Kaufmann, (**1989**)] and *An Integrated Introduction to Computer Graphics and Geometric Modelling* by Ronald Goldman [CRC Press (**2009**)].

3.4 Primitive solids

`MCS` currently supports four types of *primitive solids*: the *parallelepiped*, the *sphere*, the *cylinder*, and the *torus*.

43 ⟨Type definitions 6⟩ +≡
```
typedef enum {
  BLOCK = 0, SPHERE, CYLINDER, TORUS
} Primitive_type;
```

The standard primitives listed above are generic; solid instances of these primitives can take different shapes and form. To use solids of a given primitive type, a new instance of the generic primitive must first be created. This must then be initialised to the required specification by filling in the relevant parameters.

An instance of a primitive solid defines a convex solid volumes. A solid is convex if any line segment between any two points on the surface of the solid only intersects the solid at those two points. We represent an instance of a primitive solid using the *Primitive* data type.

NOTE: In each of the following sections, we specify only the relevant details in relation to the context of the section. Additional details will be incorporated when we discuss other aspects of the solids. For instance, at the moment we are only concerned with the geometry of the solids, and not with their material properties. Hence, the details concerning their material properties will be added later on, when we discuss materials.

44 ⟨Type definitions 6⟩ +≡
```
typedef struct {
  ⟨Information common to all primitives 45⟩;
  union {
    ⟨Data for primitive block 47⟩;
    ⟨Data for primitive sphere 49⟩;
    ⟨Data for primitive cylinder 51⟩;
    ⟨Data for primitive torus 53⟩;
  };
} Primitive;
```

The first field of all primitive data stores a primitive type. This is used while deciding the manner in which a solid instance must be processed: based on this type, we choose the appropriate C/ union field.

45 ⟨Information common to all primitives 45⟩ ≡
```
Primitive_type type;
```
This code is used in chunk 44.

In addition to the *global coordinate frame* defined by the simulation world, each initialised primitive also defines a *local coordinate frame*. The origin of this coordinate frame is used by the geometry construction algorithm to conceptually place a primitive inside the simulation world. By design, a primitive solid is created in such a way that its origin always coincides with the origin of the world coordinate frame. Any translation or transformation henceforth is recorded separately in a binary tree. This will become clearer once we reach the section on *Constructive Solid Geometry Tree*.

3.4.1 Parallelepiped

For simplicity, we shall refer to a parallelepiped as a *block*. A primitive block stores the following information.

```
47 ⟨Data for primitive block 47⟩ ≡
     struct {
       ⟨Information that defines a primitive block 48⟩;
     } b;
```

This code is used in chunk 44.

The geometry of a block is defined by its length, width and height. The origin of the block's local coordinate frame is defined by its *centroid*, and the orthogonal axes incident on this origin, i.e., the x, y and z axes of the local coordinate frame, are aligned respectively in parallel to its length, height and width. For efficient containment testing, we store half-lengths, instead of storing the full length, width, or height.

```
48 ⟨Information that defines a primitive block 48⟩ ≡
     double length, height, width;      /* half-lengths */
```

See also chunk 197.

This code is used in chunk 47.

3.4.2 Sphere

A primitive sphere stores the following information.

```
49 ⟨Data for primitive sphere 49⟩ ≡
     struct {
       ⟨Information that defines a primitive sphere 50⟩;
     } s;
```

This code is used in chunk 44.

The geometry of a sphere is defined by its radius, and the origin of its local coordinate frame is defined by the sphere's *center*.

50 ⟨Information that defines a primitive sphere 50⟩ ≡

```
  double radius;
```

This code is used in chunk 49.

3.4.3 Cylinder

A primitive cylinder stores the following information.

51 ⟨Data for primitive cylinder 51⟩ ≡

```
  struct {
     ⟨Information that defines a primitive cylinder 52⟩;
  } c;
```

This code is used in chunk 44.

The geometry of a cylinder is defined by its base radius and its height. The origin of the block's local coordinate frame is defined by its *centroid*, and the orthogonal axes incident on this origin, i.e., the x, y and z axes of the local coordinate frame, are aligned so that y axis is perpendicular to the base.

52 ⟨Information that defines a primitive cylinder 52⟩ ≡

```
  double radius, height;
```

See also chunk 201.

This code is used in chunk 51.

3.4.4 Torus

A primitive torus stores the following information.

53 ⟨Data for primitive torus 53⟩ ≡

```
  struct {
     ⟨Information that defines a primitive torus 54⟩;
  } t;
```

This code is used in chunk 44.

The origin of the torus' local coordinate frame is defined by its *center*, which coincides with the origin of the world coordinate frame. We also assume that they are radially symmetrical to the y-axis of the world coordinate frame. Under these condition, a torus may be defined parametrically as follows:

$$\begin{aligned}
x(\phi,\theta) &= (R + r\cos\theta)\cos\phi\\
y(\phi,\theta) &= (R + r\cos\theta)\sin\phi\\
z(\phi,\theta) &= r\sin\theta
\end{aligned}$$

where, the length R gives the distance from the center of the tube to the center of the torus, and the length r gives the radius of the tube. They are referred to as the *major radius* and the *minor radius*, respecticely; and their ratio is referred to as the *aspect ratio*. Parameters ϕ and θ are angles in radians, where $0 \leq \phi, \theta < 2\pi$.

Parameter ϕ is the angle subtended by the center of the tube on the xz-plane due to a counter-clockwise rotation about the y-axis, where the rotation begins at the positive x-axis. Parameter θ, on the other hand, is the angle subtended by the surface of the tube on a cross-section of the tube, where the cross section is defined by a plane perpendicular to the xz-plane which passes through the center of the torus and the center of the tube, at the point where the angle subtended by this plane on the xz-plane is ϕ radians.

54 ⟨ Information that defines a primitive torus 54 ⟩ ≡
```
  double major, minor;
  double phi, phi_start, theta, theta_start;
      /* subtended angle, and start angle (in degrees) */
```
See also chunk 205.
This code is used in chunk 53.

To create a primitive solid, we use the global memory area *mem*. This memory area stores all of the primitive solids in blocks of items of type **Primitive**. Hence, we need a pointer that points to the slot that is currently available inside the active **Primitive** block, and a pointer to check if the available slot is indeed valid. The pointers *next_primitive* and *bad_primitive* keep track of this information.

The *next_primitive* pointer is updated after the allocation of a slot from the current **Primitive** block, or when a new **Primitive** block is allocated using *mem_alloc*(n, s); the *bad_primitive* is updated only when a new block is allocated.

55 ⟨ Global variables 4 ⟩ +≡
```
  static Primitive *next_primitive = Λ;
  static Primitive *bad_primitive = Λ;
```

To store a primitive solid, we check if the available slot is valid, i.e., *next_primitive* ≠ *bad_primitive*. If valid, we use this slot; otherwise, we allocate a new block that can accommodate *primitives_per_block* primitive solids, and use the first slot in this block.

56 #define *primitives_per_block* 195 /* 16384 bytes per block */
⟨ Global functions 8 ⟩ +≡
```
  Primitive *create_primitive_solid()
  {
    Primitive *slot = next_primitive;
    if ($slot \equiv bad\_primitive$) {
        $slot = mem\_typed\_alloc(primitives\_per\_block, \textbf{Primitive}, mem\_phase\_one)$;
        if ($slot \equiv \Lambda$) return $\Lambda$;
        else {
            $next\_primitive = slot + 1$;
            $bad\_primitive = slot + primitives\_per\_block$;
        }
    }
    else $next\_primitive{+}{+}$;
    return $slot$;
}
```

3.5 Constructive Solid Geometry Tree

All solids in the simulation world are built from instances of primitive solids by using volumetric boolean operators. These operators work on the volume defined by the primitive solids. There are three volumetric boolean operators: *union*, *intersection* and *difference*. They respectively define the volumetric union, intersection and difference of two solid operands.

Further to the boolean operators, affine transformation operators are defined for translation, rotation and scaling. They allow us to change the shape, form, orientation and location of its solid operand relative to the world coordinate frame. Affine transformation operators are unary, whereas boolean operators are binary.

57
```
⟨Type definitions 6⟩ +≡
  typedef enum {
    PRIMITIVE = 0, UNION, INTERSECTION, DIFFERENCE, TRANSLATE, ROTATE,
        SCALE, PARAMETER
  } csg_node_type;
```

A CSG solid is represented by a binary tree where the root represents the solid that is being created. The primitive solids are stored at the leaves, and all of the internal nodes either represent affine transformations or boolean combinations. This tree is generally referred to as the *Constructive Solid Geometry Tree*, or *CSG Tree* for short.

The performance of many of the algorithms in the following sections can be improved significantly if each of the CSG tree nodes are enclosed inside an axis-aligned parallelipiped, commonly referred to as the solid's *bounding box*. This is defined by a pair of three dimensional coordinates (λ, τ), where λ and τ are respectively the lower and upper bounds of all the points that are inside the solid.

58
```
⟨Type definitions 6⟩ +≡
  typedef struct {
    Vector l, u;      /* lower and upper bounds */
  } BoundingBox;
```

59
```
⟨Type definitions 6⟩ +≡
  typedef struct csg_node_struct CSG_Node;
  struct csg_node_struct {
    ⟨Information common to all CSG nodes 60⟩;
    union {
      ⟨Information stored in CSG leaf node 66⟩;
      ⟨Information stored in CSG internal node 65⟩;
    };
  };
```

Each node stores a node name, that uniquely identifies the node in the CSG tree. They are supplied by the user through the geometry input file. Node names are used during registration of solids.

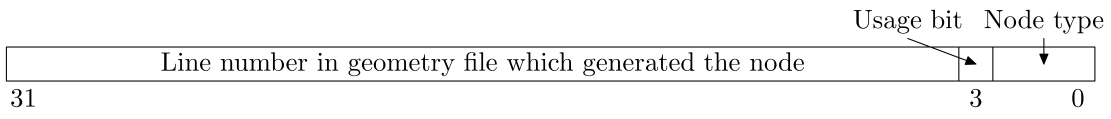
60 **#define** `MAX_LEN_NODE_NAME` 64

⟨ Information common to all CSG nodes 60 ⟩ ≡
  char *name*[`MAX_LEN_NODE_NAME`];

See also chunks 61, 64, 160, and 177.

This code is used in chunk 59.

To differentiate between node types, each node stores a CSG node type. This field determines if a node is a primitive solid node, a parameter node, or an intermediate solid node. Since we only require eight values (i.e., **csg_node_type**), it will be a waste to use more than three bits. Instead, we will use a bit field:



We make the three least significant bits to represent the node type. We are choosing the least significant bits because it will be used more often than the rest of the bit fields, which only come into effect during error diagnosis. We then use the fourth least significant bit to mark if the node is currently used in a solid CSG tree. Because the CSG tree is built bottom-up, these is a possibility that the user has made a mistake and added stray CSG nodes. Finally, the rest of the 28 bits are used to record the line number in the geometry file which gave the command to create the node. This will be displayed to the user during error diagnostics (e.g., to display a warning to the user where in the geomtry file an unused CSG node was created).

61 ⟨ Information common to all CSG nodes 60 ⟩ +≡
  uint32_t *op*;

The following macros set the bit fields.

62 **#define** `BIT_MASK_NODE` #7
#define `BIT_MASK_UBIT` #8
#define `BIT_MASK_LINE` #fffffff0
#define *set_ubit*(n) ((n)→*op* |= `BIT_MASK_UBIT`)
#define *set_line*(n) ((n)→*op* |= (*input_file_current_line* ≪ 4))
#define *get_line*(n) ((n)→*op* ≫ 4)

The following macros check the bit fields.

63 **#define** *is_inuse*(n) (`BIT_MASK_UBIT` & (n)→*op*)
#define *is_primitive*(n) (¬(`BIT_MASK_NODE` & (n)→*op*))
/∗ special because `PRIMITIVE` ≡ #`0` ∗/
#define *is_parameter*(n) (`PARAMETER` ≡ (`BIT_MASK_NODE` & (n)→*op*))
#define *is_union*(n) (`UNION` ≡ (`BIT_MASK_NODE` & (n)→*op*))
#define *is_intersection*(n) (`INTERSECTION` ≡ (`BIT_MASK_NODE` & (n)→*op*))
#define *is_difference*(n) (`DIFFERENCE` ≡ (`BIT_MASK_NODE` & (n)→*op*))
#define *is_translate*(n) (`TRANSLATE` ≡ (`BIT_MASK_NODE` & (n)→*op*))
#define *is_rotate*(n) (`ROTATE` ≡ (`BIT_MASK_NODE` & (n)→*op*))
#define *is_scale*(n) (`SCALE` ≡ (`BIT_MASK_NODE` & (n)→*op*))

All nodes store a pointer to its parent node. This is used for backtracking during tree traversal.

64 ⟨ Information common to all CSG nodes 60 ⟩ +≡
CSG_Node ∗*parent*; /∗ pointer to parent node ∗/

The internal nodes of a CSG tree store operators. If the operator stored is binary (i.e., the set-theoretic operators) both left and right children point to solids. If the operator is unary, the left child points to a solid, and the right child stores the operator parameters (e.g., the displacement if we are translating a solid, or the scaling factor if we are scaling a solid, etc.).

65 ⟨ Information stored in CSG internal node 65 ⟩ ≡
struct {
  CSG_Node ∗*left*, ∗*right*; /∗ pointers to the left and right subtrees ∗/
} *internal*;

This code is used in chunk 59.

CSG leaf nodes that point to solids, or unary operator nodes that store parameters, require additional data. A node that corresponds to a unary operator stores operator specific parameters, whereas a node that corresponds to a solid stores a pointer to a primitive solid. A leaf node, however, never points to an *intermediate solid*, which is defined as a complex solid composed of several primitive solids using a CSG tree of operators. In fact, each internal node in a CSG tree defines an intermediate solid.

66 ⟨ Information stored in CSG leaf node 66 ⟩ ≡
union {
  ⟨ Translation parameters 132 ⟩;
  ⟨ Rotation parameters 141 ⟩;
  ⟨ Scaling parameters 147 ⟩;

```
    Primitive *p;
  } leaf;
```

This code is used in chunk 59.

To create a new CSG node, we use the global memory area *mem*. This memory area stores all of the CSG nodes in blocks of items of type **CSG_Node**. Hence, we need a pointer that points to the slot that is currently available inside the active **CSG_Node** block, and a pointer to check if the available slot is indeed valid. The pointers *next_csg_node* and *bad_csg_node* keep track of this information.

The *next_csg_node* pointer is updated after the allocation of a slot from the current **CSG_Node** block, or when a new **CSG_Node** block is allocated using *mem_alloc*(n, s); the *bad_csg_node* is updated only when a new block is allocated.

```
67 ⟨ Global variables 4 ⟩ +≡
  static CSG_Node *next_csg_node = Λ;
  static CSG_Node *bad_csg_node = Λ;
```

To create a new CSG node, we check if the available slot is valid, i.e., *next_csg_node* ≠ *bad_csg_node*. If valid, we use this slot; otherwise, we allocate a new block that can accommodate *csg_node_per_block* CSG nodes, and use the first slot in this block.

```
68 #define csg_nodes_per_block 128      /* 65536 bytes per block */
⟨ Global functions 8 ⟩ +≡
  CSG_Node *create_csg_node( )
  {
    CSG_Node *slot = next_csg_node;
    if (slot ≡ bad_csg_node) {
      slot = mem_typed_alloc(csg_nodes_per_block, CSG_Node, mem_phase_one);
      if (slot ≡ Λ) return Λ;
      else {
        next_csg_node = slot + 1;
        bad_csg_node = slot + csg_nodes_per_block;
      }
    }
    else next_csg_node++;
    ⟨ Initialise affine matrices to the identity matrix 178 ⟩;
    return slot;
  }
```

3.5.1 Nodes repository

In the first phase, when the input files are processed, we must build the CSG trees which correspond to all of the solids bottom-up. This requires maintaining the

CSG nodes already defined, which may be disconnected from the others nodes, so that subsequent commands may combine them using CSG operators. Hence, we use a nodes repository to stores all of the nodes already defined. This is implemented using a hash table, where the name of the node is used as the hash key.

69
```
#define MAX_CSG_NODES 65536
        /* maximum number of CSG nodes allowed */
⟨ Type definitions 6 ⟩ +≡
  typedef struct {
    uint16_t stat[7];    /* geometry statistics */
    CSG_Node *root;    /* pointer to the CSG root */
    CSG_Node *table[MAX_CSG_NODES];    /* hash table of nodes */
  } NodesRepository;
  NodesRepository *nodes_repo = Λ;
```

The nodes repository is only used in phase one, when the data from the input files are processed at the beginning. In phase two, this information will be reduced to a compact representation which requires less memory. After phase two is complete, we must free the resources allocated to the nodes repository as it consumes quite a lot of memory. Of course, since the nodes repository is allocated in the memory area *mem_phase_one*, it will be freed automatically when we free phase one memory area at the end of phase two.

Note that the macro *mem_typed_alloc*( ) initialises all of the fields to zero, since it uses the *calloc*( ) system call to allocate the memory. Hence, we do not need to initiliase the fields separately.

70
```
⟨ Global functions 8 ⟩ +≡
  bool create_nodes_repository( )
  {
    nodes_repo = mem_typed_alloc(1, NodesRepository, mem_phase_one);
    if (Λ ≡ nodes_repo) return false;
    return true;
  }
```

Function *print_geom_statistics*(*f*) prints the geometry statistics in the nodes repository to the I/O stream pointed to by *f*. Inside the nodes repository, we only maintain the overall statistics concerning the number of primitives, operators and solids. If we wish to obtain statistics for a specific solid, it may be derived at by traversing the CSG tree that corresponds to the selected solid. The array stores in order the number of: primitives, unions, intersections, differences, translations, rotations and scalings.

71 ⟨ Global functions 8 ⟩ +≡

```
void print_geom_statistics(FILE *f)
{
  if (Λ ≡ nodes_repo) return;
  fprintf(f, "Geometry␣statistics\n\tprimitive:␣%u\n\\
      tunion:␣%u\n\tintersection:␣%u\n\tdifference:␣%u\n\ttranslat\
      ion:␣%u\n\trotation:␣%u\n\tscaling:␣%u\n", nodes_repo→stat[0],
      nodes_repo→stat[1], nodes_repo→stat[2], nodes_repo→stat[3],
      nodes_repo→stat[4], nodes_repo→stat[5], nodes_repo→stat[6]);
}
```

The hash code for a string $c_1c_2\ldots c_l$ of length l is a nonlinear function of the characters. We borrow the hash function described by Donald E. Knuth in the book *The Stanford GraphBase* [Addison-Wesley (**1993**)] to calculate h. As noted therein, this hash function only works with ASCII encoded strings.

72

```
#define HASH_MULT 314159    /* random multiplier */
#define HASH_PRIME 516595003
            /* the 27182818th prime; which is less than 2^29 */
```

⟨ Global functions 8 ⟩ +≡

```
long hash(const char *name, long ubound)
{
  register long h;
  const char *t = name;
  for (h = 0; *t; t++) {
    h += (h ⊕ (h ≫ 1)) + HASH_MULT * (unsigned char) *t;
    while (h ≥ HASH_PRIME) h −= HASH_PRIME;
  }
  return (h % ubound);
}
```

Function *register_csg_node*(n, s) registers into the nodes repository the CSG node pointed to by n using the unique name s. If any of the inputs are invalid, or if the supplied name is already in use, it returns *false* to notify registration failure; otherwise, it returns *true* to indicate success.

73 ⟨ Global functions 8 ⟩ +≡

```
bool register_csg_node(CSG_Node *n, char *s)
{
  long h;
  if (Λ ≡ n ∨ Λ ≡ s) return false;
  h = hash(s, MAX_CSG_NODES);
```

```
    if (Λ ≡ nodes_repo→table[h]) {
      strcpy(n→name, s);
      nodes_repo→table[h] = n;
      nodes_repo→root = n;      /* set as current root */
      return true;
    }
    return false;
  }
```

Function *find_csg_node*(s) finds in the nodes repository the CSG node named s. If the input is invalid, or the node does not exists, NULL is returned; otherwise, a pointer to the node is returned.

74 ⟨ Global functions 8 ⟩ +≡

```
  CSG_Node *find_csg_node(char *s)
  {
    if (Λ ≡ s) return Λ;
    return nodes_repo→table[hash(s, MAX_CSG_NODES)];
  }
```

3.5.2 The geometry input file

The geometry of the solids and their placement and orientation within the world is specified in the input file. The grammar for this input file is very simple. The file consist of several commands, where an entire line of text is used to specify a specific command. Each command has the following format:

⟨*command*⟩⟨*parameters*⟩⟨*newline*⟩

Here, ⟨*command*⟩ is a single character code, which defines its intended action. The commands and their intended actions are as follows:

`B, S, C, T`	Create a primitive block, sphere, cylinder, or torus.
`u, i, d`	Carry out a union, intersection, or difference.
`t, r, s`	Translate, rotate, or scale the solid.
`+`	Registers the current CSG tree as solid.
`*`	Defines the simulation world as an axis-aligned bounding box.
`%`	Begin comment line. Stop at the first newline character.

To support this intended action, the user must supply all of the required parameters in the ⟨*parameters*⟩ field. Finally, every command must be terminated by a ⟨*newline*⟩ character. The following is an example:

```
% Define primitive solids
```

```
   T ("Torus A" 0.0 359.999999 10.0 2.0)
   C ("Cylinder A" 10.0 20.0)
   C ("Cylinder B" 10.0 20.0)

   % Operation on primitive solids
   u ("U1" "Torus A" "Cylinder A")
   d ("D1" "U1" "Cylinder B")
   t ("T1" "D1" 10.0 50.0 20.0)
   s ("S1" "T1" 10.0 20.0 30.0)
   + ("SOLID") % register CSG tree as a solid
   * (-100.0 -100.0 -100.0 100.0 100.0 100.0) % define the simulation
world
```

Function *read_geometry*(n) reads the geometry data by opening the file named n.

```
76 ⟨ Global functions 8 ⟩ +≡
   bool read_geometry(const char *n)
   {
      ⟨ Local variables: read_geometry( ) 77 ⟩;
      ⟨ Open input geometry file and initialise nodes repository 78 ⟩;
      while (EOF ≠ (c = fgetc(f))) {
         ⟨ Discard comments, white spaces and empty lines 79 ⟩;
         ⟨ Process input command 83 ⟩;
      }
      fclose(f);
      return true;
      ⟨ Handle geometry file errors 84 ⟩;
      return false;
   }
```

```
77 ⟨ Local variables: read_geometry( ) 77 ⟩ ≡
   FILE *f;
   char c;
```

See also chunk 223.

This code is used in chunk 76.

We use the temporary variable *input_file_name* to store the name of the file that is currently being processed, so that if we wish, we might generate the name of the file at runtime (e.g., by appending prefix and suffix strings to a base filename).

78 ⟨ Open input geometry file and initialise nodes repository 78 ⟩ ≡
```
input_file_current_line = 1;
strcpy(input_file_name, n);      /* generate filename */
f = fopen(input_file_name, "r");
if (Λ ≡ f) {
  fprintf(stderr, "Failed␣to␣open␣input␣geometry␣file:␣%s\n",
       input_file_name);
  return 1;
}
if (false ≡ create_nodes_repository()) {
  fprintf(stderr, "Failed␣to␣initialise␣nodes␣repository\n");
  return 1;
}
```
This code is used in chunk 76.

To improve readability, we allow comments, empty lines and indentation of commands.

79 ⟨ Discard comments, white spaces and empty lines 79 ⟩ ≡
```
⟨ Discard comment lines 80 ⟩;
⟨ Discard indentations 81 ⟩;
⟨ Discard empty lines 82 ⟩;
```
This code is used in chunks 76 and 226.

Comments begin with the ‘%’ character, and end after the end-of-line character.

80 ⟨ Discard comment lines 80 ⟩ ≡
```
if (c ≡ '%') {
  while (EOF ≠ (c = fgetc(f)))
    if ('\n' ≡ c) break;      /* gobble comments */
  if (EOF ≡ c) break;      /* done reading input file */
}
```
This code is used in chunk 79.

81 ⟨ Discard indentations 81 ⟩ ≡
```
if ('␣' ≡ c ∨ '\t' ≡ c) continue;
```
This code is used in chunk 79.

82 ⟨ Discard empty lines 82 ⟩ ≡
```
if ('\n' ≡ c) {
  ++input_file_current_line;
  continue;
}
```
This code is used in chunk 79.

83 ⟨ Process input command 83 ⟩ ≡
```
switch (c) {
case 'B': ⟨ Read block geometry 94 ⟩; break;
case 'S': ⟨ Read sphere geometry 103 ⟩; break;
case 'C': ⟨ Read cylinder geometry 105 ⟩; break;
case 'T': ⟨ Read torus geometry 109 ⟩; break;
case 'u': ⟨ Read union operation 113 ⟩; break;
case 'i': ⟨ Read intersection operation 125 ⟩; break;
case 'd': ⟨ Read difference operation 129 ⟩; break;
case 't': ⟨ Read translation operation 133 ⟩; break;
case 'r': ⟨ Read rotation operation 142 ⟩; break;
case 's': ⟨ Read scaling operation 148 ⟩; break;
case '+': ⟨ Read solid registration 153 ⟩; break;
case '*': ⟨ Read simulation world specification 157 ⟩; break;
default:
  fprintf(stderr, "%s[%u]␣Invalid␣command␣'%c'␣in␣input␣file\n",
      input_file_name, input_file_current_line, c);
  goto error_invalid_file;
}
```
This code is used in chunk 76.

When we cannot recover from an error (e.g., incorrect input file), we must exit the system after cleaning up the resources that were allocated by previous commands. Furthermore, the system must also alert the user about the error. This section defines all of the exit points and the corresponding error messages.

84 ⟨ Handle geometry file errors 84 ⟩ ≡
```
  ⟨ Alert error while reading file 85 ⟩;
  ⟨ Alert failure to create primitive solid 86 ⟩;
  ⟨ Alert failure to create operator node 87 ⟩;
  ⟨ Alert failure to create parameter node 88 ⟩;
  ⟨ Alert solid already exists 89 ⟩;
  ⟨ Alert solid does not exists 90 ⟩;
  ⟨ Alert invalid division of simulation world 91 ⟩;
error_invalid_file:
  ⟨ Cleanup resources allocated to invalid geometry 92 ⟩;
  fclose(f);
```
This code is used in chunk 76.

85 ⟨ Alert error while reading file 85 ⟩ ≡

```
failed_read_exit_after_cleanup:
   fprintf(stderr,
        "Failed␣to␣read␣file␣'%s'␣at␣line␣%u\n""\tInvalid␣formattin\
        g␣of␣parameters\n", input_file_name, input_file_current_line);
   goto error_invalid_file;
```

This code is used in chunk 84.

86 ⟨ Alert failure to create primitive solid 86 ⟩ ≡

```
create_primitive_failed_exit_after_cleanup:
   fprintf(stderr, "%s[%u]␣Failed␣to␣create␣primitive␣solid\n",
        input_file_name, input_file_current_line);
   goto error_invalid_file;
```

This code is used in chunk 84.

87 ⟨ Alert failure to create operator node 87 ⟩ ≡

```
create_operator_failed_exit_after_cleanup:
   fprintf(stderr, "%s[%u]␣failed␣to␣create␣internal␣node\n",
        input_file_name, input_file_current_line);
   goto error_invalid_file;
```

This code is used in chunk 84.

88 ⟨ Alert failure to create parameter node 88 ⟩ ≡

```
create_parameter_failed_exit_after_cleanup:
   fprintf(stderr, "%s[%u]␣failed␣to␣create␣leaf␣node\n", input_file_name,
        input_file_current_line);
   goto error_invalid_file;
```

This code is used in chunk 84.

89 ⟨ Alert solid already exists 89 ⟩ ≡

```
solid_exists_exit_after_cleanup:
   fprintf(stderr, "%s[%u]␣Invalid␣geometry␣specification...␣"
   "Solid␣named␣'%s'␣already␣exists\n", input_file_name,
        input_file_current_line, solid_name);
   goto error_invalid_file;
```

This code is used in chunk 84.

```
90 ⟨ Alert solid does not exists 90 ⟩ ≡
no_solid_exists_exit_after_cleanup:
  fprintf(stderr, "%s[%u]␣Invalid␣geometry␣specification...␣"
  "Solid␣named␣'%s'␣does␣not␣exists\n", input_file_name,
      input_file_current_line, solid_name);
  goto error_invalid_file;
```
This code is used in chunk 84.

```
91 ⟨ Alert invalid division of simulation world 91 ⟩ ≡
invalid_subcuboid_division_exit_after_cleanup:
  fprintf(stderr, "%s[%u]␣Invalid␣geometry␣specification...␣"
  "Number␣of␣subcuboids␣%u␣exceeds␣allowed␣%u\n", input_file_name,
      input_file_current_line, num_subcuboids, MAX_SUBCUBOIDS);
  goto error_invalid_file;
```
This code is used in chunk 84.

```
92 ⟨ Cleanup resources allocated to invalid geometry 92 ⟩ ≡
  fprintf(stderr, "Cleaning␣up␣resources...\n");
```
This code is used in chunk 84.

While reading in the operations, we use the following variables to store temporary values. Since each operation is defined independently of other operations before and after, these variables can be shared by all of the operation commands without confusion.

```
93 #define MAX_LEN_FILENAME 256
⟨ Global variables 4 ⟩ +≡
  double op_x, op_y, op_z, op_theta;
  char op_target[MAX_LEN_NODE_NAME], op_solid[MAX_LEN_NODE_NAME];
  char op_left[MAX_LEN_NODE_NAME], op_right[MAX_LEN_NODE_NAME];
  char *solid_name;      /* points to name of solid while alerting error */
  CSG_Node *target_solid, *left_solid, *right_solid;
```

The parameters that are required to initialise a standard primitive is read from an input file. These parameters must be supplied using specific formatting requirements. The parameters for the block geometry are supplied in the following format.

(“name” *length width height*)

For instance, the specification `("Block A" 10.0 5.0 15.0)` initialises a new block named “Block A” with centroid at (0.0, 0.0, 0.0) in the world coordinate frame with length 10.0, width 5.0 and height 15.0. It is important that all of these lengths are specified using the same unit of measurement. We shall discuss in later sections how a unit of measurement is chosen.

```
94 ⟨ Read block geometry 94 ⟩ ≡
     ⟨ Create a new primitive solid 95 ⟩;
     ⟨ Initialise primitive block with relevant data 98 ⟩;
     ⟨ Register the primitive solid 102 ⟩;
```

This code is used in chunk 83.

```
95 ⟨ Create a new primitive solid 95 ⟩ ≡
     p = create_primitive_solid( );
     if (Λ ≡ p) {
        ⟨ Exit after cleanup: failed to create primitive solid 96 ⟩;
     }
```

This code is used in chunks 94, 103, 105, and 109.

To increase clarity and to reduce code size, we keep recurring error messages in common code-blocks, and jump to these blocks as required. When it comes to implementing error handling, such as this, where we wish to terminate the application after cleanup, I prefer `goto` statements. This is where bending the rules of *Structured Programming* leads to cleaner source code and smaller object code.

```
96 ⟨ Exit after cleanup: failed to create primitive solid 96 ⟩ ≡
     goto create_primitive_failed_exit_after_cleanup;
```

This code is used in chunk 95.

We use the following variables while reading input data from a file.

```
97 ⟨ Global variables 4 ⟩ +≡
     char input_file_name[MAX_LEN_FILENAME];    /* current input file */
     uint32_t input_file_current_line;    /* current line in current input file */
     int read_count;    /* returned by fscanf() */
     Primitive *p;    /* used during initialisation of primitive solids */
     CSG_Node *internal_node, *leaf_node, *temp_node;
```

```
98 ⟨ Initialise primitive block with relevant data 98 ⟩ ≡
     read_count = fscanf(f, "(\"%[^\"]\"␣%lf␣%lf␣%lf)", op_solid, &p→b.length,
           &p→b.width, &p→b.height);
     if (EOF ≡ read_count ∨ 4 ≠ read_count) {
        ⟨ Exit after cleanup: failed to read from file 101 ⟩;
}
p→type = BLOCK;
⟨ Prepare block for containment testing 99 ⟩;
++(nodes_repo→stat[PRIMITIVE]);
```

This code is used in chunk 94.

To improve containment testing, we precalculate some of the values that are used frequently. We first half the dimensions, and then calculate the containment range for the block.

99 ⟨ Prepare block for containment testing 99 ⟩ ≡

```
⟨ Half the length, height and width of the block 100 ⟩;
⟨ Calculate containment range for the block 198 ⟩;
```

This code is used in chunk 98.

When checking containment, we require the halves of the block length, height and width. Hence, although the dimensions were specified in full, we shall store their halves internally.

100 ⟨ Half the length, height and width of the block 100 ⟩ ≡

```
p→b.length /= 2.0;
p→b.height /= 2.0;
p→b.width /= 2.0;
```

This code is used in chunk 99.

101 ⟨ Exit after cleanup: failed to read from file 101 ⟩ ≡

```
goto failed_read_exit_after_cleanup;
```

This code is used in chunks 98, 104, 106, 110, 113, 125, 129, 134, 143, 149, 154, and 157.

102 ⟨ Register the primitive solid 102 ⟩ ≡

```
leaf_node = create_csg_node( );
if (Λ ≡ leaf_node) {
  ⟨ Exit after cleanup: failed to create leaf node 137 ⟩;
}
else {
  leaf_node→op = PRIMITIVE;     /* this is a primitive solid leaf node */
  leaf_node→leaf.p = p;
  leaf_node→parent = Λ;
  set_line(leaf_node);
  register_csg_node(leaf_node, op_solid);
}
```

This code is used in chunks 94, 103, 105, and 109.

The parameters for the sphere geometry are supplied in the following format:

(“name” *radius*)

For instance, the specification `("Sphere A" 10.0)` initialises a new solid sphere named “Sphere A” located at (0.0, 0.0, 0.0) in the world coordinate frame with a radius of 10.0. Again, we assume that the radius is specified in the chosen unit of measurement, yet to be discussed.

```
103 ⟨Read sphere geometry 103⟩ ≡
      ⟨Create a new primitive solid 95⟩;
      ⟨Initialise primitive sphere with relevant data 104⟩;
      ⟨Register the primitive solid 102⟩;
```

This code is used in chunk 83.

```
104 ⟨Initialise primitive sphere with relevant data 104⟩ ≡
      read_count = fscanf(f, "(\"%[^\"]\"␣%lf)", op_solid, &p→s.radius);
      if (EOF ≡ read_count ∨ 2 ≠ read_count) {
         ⟨Exit after cleanup: failed to read from file 101⟩;
      }
      p→type = SPHERE;
      ++(nodes_repo→stat[PRIMITIVE]);
```

This code is used in chunk 103.

The parameters for the cylinder geometry are supplied in the following format:

(“name” *radius height*)

For instance, the specification `("Cylinder A" 10.0 20.0)` initialises a new solid cylinder named “Cylinder A” located at (0.0, 0.0, 0.0) in the world coordinate frame with base radius 10.0 and height 20.0.

```
105 ⟨Read cylinder geometry 105⟩ ≡
      ⟨Create a new primitive solid 95⟩;
      ⟨Initialise primitive cylinder with relevant data 106⟩;
      ⟨Register the primitive solid 102⟩;
```

This code is used in chunk 83.

```
106 ⟨Initialise primitive cylinder with relevant data 106⟩ ≡
      read_count = fscanf(f, "(\"%[^\"]\"␣%lf␣%lf)", op_solid, &p→c.radius,
            &p→c.height);
      if (EOF ≡ read_count ∨ 3 ≠ read_count) {
         ⟨Exit after cleanup: failed to read from file 101⟩;
      }
      p→type = CYLINDER;
```

⟨ Prepare cylinder for containment testing 107 ⟩;
++(*nodes_repo*→*stat*[PRIMITIVE]);
This code is used in chunk 105.

107 ⟨ Prepare cylinder for containment testing 107 ⟩ ≡
⟨ Half the height the cylinder 108 ⟩;
⟨ Calculate containment range for the cylinder 202 ⟩;
This code is used in chunk 106.

When checking containment, we require half of the cylinder *height*, because the origin of the cylinder is given by its centroid. Hence, although the *height* were specified in full, we shall store their halves internally.

108 ⟨ Half the height the cylinder 108 ⟩ ≡
p→*c.height* /= 2.0;
This code is used in chunk 107.

The parameters for the torus geometry are supplied in the following format:

("name" *phi phi_start theta theta_start major minor*)

For instance, the specification `("Torus A" 180.0 45.0 45.0 15.0 10.0 2.0)` initialises a partial torus named "Torus A" located at (0.0, 0.0, 0.0) in the world coordinate frame with major radius 10.0 and minor 2.0. Its start and end cross sections subtend an angle of 180°, and begins it rotation at 45° from the positive x-axis. Furthermore, the tube subtends and angle of 45°, and begins its rotation about the center of the tube starting at 15° from the xz-plane.

Note here that, although mathematically the parametric angles $\phi, \theta < 2\pi$ are defined as radians, their computational values *phi* and *theta* (and their start angles *phi_start* and *theta_start*) are supplied in degrees. This avoids conversion between radians and degrees while carrying out containment testing. Furthermore, we allow variables *phi* and *theta* to take values of 360°, which signifie complete rotations.

109 ⟨ Read torus geometry 109 ⟩ ≡
⟨ Create a new primitive solid 95 ⟩;
⟨ Initialise primitive torus with relevant data 110 ⟩;
⟨ Register the primitive solid 102 ⟩;
This code is used in chunk 83.

```
110 ⟨ Initialise primitive torus with relevant data 110 ⟩ ≡
    read_count = fscanf(f, "(\"%[^\"]\"␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf)",
        op_solid, &p→t.phi, &p→t.phi_start, &p→t.theta, &p→t.theta_start,
        &p→t.major, &p→t.minor);
    if (EOF ≡ read_count ∨ 7 ≠ read_count) {
      ⟨ Exit after cleanup: failed to read from file 101 ⟩;
    }
    p→type = TORUS;
    ⟨ Prepare torus for containment testing 111 ⟩;
    ++(nodes_repo→stat[PRIMITIVE]);
```
This code is used in chunk 109.

```
111 ⟨ Prepare torus for containment testing 111 ⟩ ≡
    ⟨ Calculate radial containment range for the torus 206 ⟩;
    ⟨ Calculate end angles for the torus 207 ⟩;
```
This code is used in chunk 110.

3.5.3 Union

The union of two solids is defined as the volume which consist of points that are **inside either of** these solids. In the geometry input file, the union of two solids are defined using the following format:

(“target” “left solid” “right solid”)

Here, “left solid” and “right solid” are the names of the union operands, and the result of the union is to be stored using the name “target”. For the union command to be valid, no solid with the name “target” must already exists within the system. Whereas, both operands must already exists within the system and could represent primitive solids, or intermediate solids that are defined by a CSG sub-tree. They are not required to share space (i.e., the solids may be detached from one another). Because the union operator is *commutative*, the order of the operands is unimportant. However, for performance considerations, it is advisable to order the solids so that the solid on the left requires the least amount of computation to determine inclusion (i.e., determination of whether a point exists inside the solid).

For instance, the union specification `("U1" "Cylinder A" "Torus A")` finds the union of two solids named “Cylinder A” and “Torus A” and stores the result using the name “U1”.

113 ⟨Read union operation 113⟩ ≡
```
read_count = fscanf(f, "(\"%[^\"]\"␣\"%[^\"]\"␣\"%[^\"]\")", op_target,
        op_left, op_right);
if (EOF ≡ read_count ∨ 3 ≠ read_count)
    ⟨Exit after cleanup: failed to read from file 101⟩;
⟨Check that the target does not already exists 114⟩;
⟨Create union operator node 116⟩;
```
This code is used in chunk 83.

114 ⟨Check that the target does not already exists 114⟩ ≡
```
target_solid = find_csg_node(op_target);
if (Λ ≠ target_solid) {
    solid_name = op_target;
    ⟨Exit after cleanup: solid already exists 115⟩;
}
```
This code is used in chunks 113, 125, and 129.

115 ⟨Exit after cleanup: solid already exists 115⟩ ≡
```
goto solid_exists_exit_after_cleanup;
```
This code is used in chunk 114.

We are now ready to create a union operator. But first, we must ensure that the solids specified as the operands already exists within the system. If they are, we create a new operator node and make its left and right subtrees point to these existing solids.

116 ⟨Create union operator node 116⟩ ≡
```
⟨Find solid that corresponds to the left-hand operand 117⟩;
⟨Find solid that corresponds to the right-hand operand 119⟩;
⟨Create new union operator node 120⟩;
```
This code is used in chunk 113.

117 ⟨Find solid that corresponds to the left-hand operand 117⟩ ≡
```
left_solid = find_csg_node(op_left);
if (Λ ≡ left_solid) {
    solid_name = op_left;
    ⟨Exit after cleanup: solid does not exists 118⟩;
}
```
This code is used in chunks 116, 126, and 130.

118 ⟨ Exit after cleanup: solid does not exists 118 ⟩ ≡

```
goto no_solid_exists_exit_after_cleanup;
```

This code is used in chunks 117, 119, 135, 144, and 150.

119 ⟨ Find solid that corresponds to the right-hand operand 119 ⟩ ≡

```
right_solid = find_csg_node(op_right);
if (Λ ≡ right_solid) {
   solid_name = op_right;
   ⟨ Exit after cleanup: solid does not exists 118 ⟩;
}
```

This code is used in chunks 116, 126, and 130.

When a new union operator node is created, we are actually creating a new *intermediate solid*. Hence, this new solid must be registered with the system, so that they may be used by later commands.

120 ⟨ Create new union operator node 120 ⟩ ≡

```
internal_node = create_csg_node( );
if (Λ ≡ internal_node) {
   ⟨ Exit after cleanup: failed to create internal node 123 ⟩;
}
else {
   internal_node→op = UNION;      /* this is an internal node */
   ⟨ Set line number of node and usage bits of children 121 ⟩;
   ⟨ Set parents, and pointers to intermediate solids 122 ⟩;
   register_csg_node(internal_node, op_target);
      /* register operator node using the target name */
   ++(nodes_repo→stat[UNION]);
}
```

This code is used in chunk 116.

121 ⟨ Set line number of node and usage bits of children 121 ⟩ ≡

```
set_line(internal_node);
set_ubit(left_solid);
set_ubit(right_solid);
```

This code is used in chunks 120, 127, and 131.

122 ⟨ Set parents, and pointers to intermediate solids 122 ⟩ ≡

```
internal_node→internal.left = left_solid;
internal_node→internal.right = right_solid;
internal_node→parent = Λ;
```

```
left_solid→parent = internal_node;
right_solid→parent = internal_node;
```

This code is used in chunks 120, 127, and 131.

123 ⟨Exit after cleanup: failed to create internal node 123⟩ ≡

```
goto create_operator_failed_exit_after_cleanup;
```

This code is used in chunks 120, 127, 131, 138, 146, and 152.

3.5.4 Intersection

The intersection of two solids is defined as the volume which consist of points that are **both inside** these solids. In the geometry input file, the intersection is specified using the same format as that of union:

(“target” “left solid” “right solid”)

Here, “left solid” and “right solid” are the names of the intersection operands, and the result of the intersection is to be stored using the name “target”. For the intersection command to be valid, no solid with the name “target” must already exists within the system. Whereas, both operands must already exists within the system and could represent primitive solids, or intermediate solids that are defined by a CSG sub-tree. Because the intersection operator is *commutative*, the order of the operands is unimportant.

For instance, the intersection specification `("I1" "Cylinder A" "Torus A")` finds the intersection of two solids named “Cylinder A” and “Torus A” and stores the result using the name “I1”.

125 ⟨Read intersection operation 125⟩ ≡

```
read_count = fscanf(f, "(\"%[^\"]\"␣\"%[^\"]\"␣\"%[^\"]\")", op_target,
      op_left, op_right);
if (EOF ≡ read_count ∨ 3 ≠ read_count)
   ⟨Exit after cleanup: failed to read from file 101⟩;
⟨Check that the target does not already exists 114⟩;
⟨Create intersection operator node 126⟩;
```

This code is used in chunk 83.

We are now ready to create a intersection operator. But first, we must ensure that the solids specified as the operands already exists within the system. If they are, we create a new operator node and make its left and right subtrees point to these existing solids.

```
126 ⟨ Create intersection operator node 126 ⟩ ≡
      ⟨ Find solid that corresponds to the left-hand operand 117 ⟩;
      ⟨ Find solid that corresponds to the right-hand operand 119 ⟩;
      ⟨ Create new intersection operator node 127 ⟩;
```

This code is used in chunk 125.

When a new intersection operator node is created, we are actually creating a new *intermediate solid*. Hence, this new solid must be registered with the system, so that they may be used by later commands.

```
127 ⟨ Create new intersection operator node 127 ⟩ ≡
      internal_node = create_csg_node( );
      if (Λ ≡ internal_node) {
         ⟨ Exit after cleanup: failed to create internal node 123 ⟩;
      }
      else {
         internal_node→op = INTERSECTION;      /* this is an internal node */
         ⟨ Set line number of node and usage bits of children 121 ⟩;
         ⟨ Set parents, and pointers to intermediate solids 122 ⟩;
         register_csg_node(internal_node, op_target);
             /* register operator node using the target name */
         ++(nodes_repo→stat[INTERSECTION]);
      }
```

This code is used in chunk 126.

3.5.5 Difference

The difference between two solids is defined as the volume which consist of points that are **inside the first** solid, but **not inside the second**. In the geometry input file, the difference is specified using the same format as that of union and intersection:

(“target” “left solid” “right solid”)

Here, “left solid” and “right solid” are the names of the operands, and the difference is to be stored using the name “target”. For the difference command to be valid, no solid with the name “target” must already exists within the system. Whereas, both operands must already exists within the system and could represent primitive solids, or intermediate solids that are defined by a CSG sub-tree. Because the difference operator is *non-commutative*, the order of the operands is important. Hence, the difference consists of all the points that are inside the left-hand operand, but not inside the right-hand operand.

For instance, the difference specification `("D1" "Cylinder A" "Torus A")` finds the difference by subtracting “Torus A” from “Cylinder A”, and stores the result using the name “D1”.

```
129 ⟨ Read difference operation 129 ⟩ ≡
    read_count = fscanf(f, "(\"%[^\"]\"␣\"%[^\"]\"␣\"%[^\"]\")", op_target,
        op_left, op_right);
    if (EOF ≡ read_count ∨ 3 ≠ read_count)
      ⟨ Exit after cleanup: failed to read from file 101 ⟩;
    ⟨ Check that the target does not already exists 114 ⟩;
    ⟨ Create difference operator node 130 ⟩;
```

This code is used in chunk 83.

We are now ready to create a difference operator. But first, we must ensure that the solids specified as the operands already exists within the system. If they are, we create a new operator node and make its left and right subtrees point to these existing solids.

```
130 ⟨ Create difference operator node 130 ⟩ ≡
    ⟨ Find solid that corresponds to the left-hand operand 117 ⟩;
    ⟨ Find solid that corresponds to the right-hand operand 119 ⟩;
    ⟨ Create new difference operator node 131 ⟩;
```

This code is used in chunk 129.

When a new difference operator node is created, we are actually creating a new *intermediate solid*. Hence, this new solid must be registered with the system, so that they may be used by later commands.

```
131 ⟨ Create new difference operator node 131 ⟩ ≡
    internal_node = create_csg_node( );
    if (Λ ≡ internal_node) {
      ⟨ Exit after cleanup: failed to create internal node 123 ⟩;
    }
    else {
      internal_node→op = DIFFERENCE;      /* this is an internal node */
      ⟨ Set line number of node and usage bits of children 121 ⟩;
      ⟨ Set parents, and pointers to intermediate solids 122 ⟩;
      register_csg_node(internal_node, op_target);
          /* register operator node using the target name */
      ++(nodes_repo→stat[DIFFERENCE]);
    }
```

This code is used in chunk 130.

3.5.6 Translation

Translation relocates a solid by displacing the solid's origin in the x, y and z axes with respect to the world coordinate frame. The paramaters for a translation operator is specified as a *displacement* vector. The unit of displacement depends on the unit chosen by the user when specifying the CSG tree; however, this unit does not matter within the system since all measurement units are normalised internally, so that all measurements in a given category can be combined readily with other measurements in the same category.

132 ⟨ Translation parameters 132 ⟩ ≡
```
  struct {
    Vector displacement;      /* displacement of solid's origin */
  } t;
```
This code is used in chunk 66.

133 ⟨ Read translation operation 133 ⟩ ≡
```
  ⟨ Read translation parameters 134 ⟩;
  ⟨ Find the target solid for the operation 135 ⟩;
  ⟨ Create translation parameter node 136 ⟩;
  ⟨ Create and register translation operator node 138 ⟩;
```
This code is used in chunk 83.

The translation parameters are specified in the geometry input file using the following format:

("target" "solid" *dx dy dz*)

where, "target" is the name of the target solid which is obtained by translating "solid". The lengths *dx*, *dy* and *dz* are the respective displacements along the *x*, *y* and *z* axes in the world coordinate frame. For instance, the translation specification `("T1" "Solid" 10.0 50.0 20.0)` means, translate the solid associated with the name "Solid" by displacing its origin by 10.0 units along the *x* axis, 50.0 units along the *y* axis, and 20.0 units along the *z* axis, and register this intermediate solid as "T1".

134 ⟨ Read translation parameters 134 ⟩ ≡
```
  read_count = fscanf(f, "(\"%[^\"]\"␣\"%[^\"]\"␣%lf␣%lf␣%lf)", op_target,
        op_solid, &op_x, &op_y, &op_z);
  if (EOF ≡ read_count ∨ 5 ≠ read_count)
    ⟨ Exit after cleanup: failed to read from file 101 ⟩;
```
This code is used in chunk 133.

In order for a translation to be applicable, the supplied target solid must already exists within the system, either by prior definition as a primitive solid, or as an intermediate solid defined by a CSG subtree.

```
135 ⟨ Find the target solid for the operation 135 ⟩ ≡
      target_solid = find_csg_node(op_solid);
      if (Λ ≡ target_solid) {
        solid_name = op_solid;
        ⟨ Exit after cleanup: solid does not exists 118 ⟩;
      }
```

See also chunks 144 and 150.

This code is used in chunks 133, 142, 148, and 153.

We now have a valid translation, so create the parameter leaf node which will become the right-child of the translation operator node.

```
136 ⟨ Create translation parameter node 136 ⟩ ≡
      leaf_node = create_csg_node( );
      if (Λ ≡ leaf_node) {
        ⟨ Exit after cleanup: failed to create leaf node 137 ⟩;
      }
      else {
        leaf_node→op = PARAMETER;      /* this is a parameter leaf node */
        leaf_node→leaf.t.displacement[0] = op_x;
        leaf_node→leaf.t.displacement[1] = op_y;
        leaf_node→leaf.t.displacement[2] = op_z;
        ⟨ Set up the matrix for translation 222 ⟩;
      }
```

This code is used in chunk 133.

```
137 ⟨ Exit after cleanup: failed to create leaf node 137 ⟩ ≡
      goto create_parameter_failed_exit_after_cleanup;
```

This code is used in chunks 102, 136, 145, and 151.

Finally, create the translation operator node, and attach the target solid and the parameter node.

```
138 ⟨ Create and register translation operator node 138 ⟩ ≡
      internal_node = create_csg_node( );
      if (Λ ≡ internal_node) {
        ⟨ Exit after cleanup: failed to create internal node 123 ⟩;
      }
      else {
        internal_node→op = TRANSLATE;      /* this is an operator internal node */
        ⟨ Set line number of node and usage bit of transformed node 139 ⟩;
        ⟨ Set target, parameters and parent pointers 140 ⟩;
```

```
    register_csg_node(internal_node, op_target);
    ++(nodes_repo→stat[TRANSLATE]);
  }
```
This code is used in chunk 133.

139 ⟨ Set line number of node and usage bit of transformed node 139 ⟩ ≡
```
  set_line(internal_node);
  set_ubit(target_solid);
```
This code is used in chunks 138, 146, and 152.

140 ⟨ Set target, parameters and parent pointers 140 ⟩ ≡
```
  internal_node→internal.left = target_solid;
  internal_node→internal.right = leaf_node;
  internal_node→parent = Λ;
  target_solid→parent = internal_node;
  leaf_node→parent = internal_node;
```
This code is used in chunks 138, 146, and 152.

3.5.7 Rotation

We rotate a solid relative to an axis. The angle of rotation θ is specified in *radians*. If $\theta < 0$, we have *clockwise rotation*; if $\theta > 0$, we have *counter-clockwise rotation*. When $\theta = 0$, no rotation is applied. In order to apply a rotation, we also need an *axis of rotation*. This is specified as a unit vector named *axis*, which is defined relative to the origin of the world coordinate frame.

141 ⟨ Rotation parameters 141 ⟩ ≡
```
  struct {
    double theta;      /* angle of rotation in radians */
    Vector axis;
       /* axis of rotation (unit vector from origin of world coordinate frame) */
  } r;
```
This code is used in chunk 66.

142 ⟨ Read rotation operation 142 ⟩ ≡
```
  ⟨ Read rotation parameters 143 ⟩;
  ⟨ Find the target solid for the operation 135 ⟩;
  ⟨ Create rotation parameter node 145 ⟩;
  ⟨ Create and register rotation operator node 146 ⟩;
```
This code is used in chunk 83.

The rotation parameters are specified in the geometry input file using the following format:

(“target” “solid” θ *ux* *uy* *uz*)

where, “target” is the name of the solid which is obtained by rotating “solid” by an angle of θ radians relative to the axis defined by the unit vector with components *ux*, *uy* and *uz* respective along the x, y and z axes. For instance, the rotation specification `("R1" "Solid" 90.0 1.0 0.0 0.0)` means, rotate the solid associated with the name “Solid” by 90.0 degree radians relative to the x-axis in world coordinate frame and register this intermediate solid as “R1”.

143 ⟨ Read rotation parameters 143 ⟩ ≡
```
read_count = fscanf(f, "(\"%[^\"]\"␣\"%[^\"]\"␣%lf␣%lf␣%lf␣%lf)",
      op_target, op_solid, &op_theta, &op_x, &op_y, &op_z);
if (EOF ≡ read_count ∨ 6 ≠ read_count)
  ⟨ Exit after cleanup: failed to read from file 101 ⟩;
```
This code is used in chunk 142.

In order for a rotation to be applicable, the supplied target solid must already exists within the system, either by prior definition as a primitive solid, or as an intermediate solid defined by a CSG subtree.

144 ⟨ Find the target solid for the operation 135 ⟩ +≡
```
target_solid = find_csg_node(op_solid);
if (Λ ≡ target_solid) {
  solid_name = op_solid;
  ⟨ Exit after cleanup: solid does not exists 118 ⟩;
}
```

We now have a valid rotation, so create the parameter leaf node.

145 ⟨ Create rotation parameter node 145 ⟩ ≡
```
leaf_node = create_csg_node( );
if (Λ ≡ leaf_node) {
  ⟨ Exit after cleanup: failed to create leaf node 137 ⟩;
}
else {
  leaf_node→op = PARAMETER;      /* this is a parameter leaf node */
  leaf_node→leaf.r.theta = op_theta;      /* angle of rotation */
  leaf_node→leaf.r.axis[0] = op_x;
  leaf_node→leaf.r.axis[1] = op_y;
  leaf_node→leaf.r.axis[2] = op_z;
  ⟨ Set up the matrix for rotation 224 ⟩;
}
```
This code is used in chunk 142.

Finally, create the rotation operator node, and attach the target solid and the parameter node.

```
146 ⟨Create and register rotation operator node 146⟩ ≡
      internal_node = create_csg_node();
      if (Λ ≡ internal_node) {
         ⟨Exit after cleanup: failed to create internal node 123⟩;
      }
      else {
         internal_node→op = ROTATE;     /* this is an operator internal node */
         ⟨Set line number of node and usage bit of transformed node 139⟩;
         ⟨Set target, parameters and parent pointers 140⟩;
         register_csg_node(internal_node, op_target);
         ++(nodes_repo→stat[ROTATE]);
      }
```

This code is used in chunk 142.

3.5.8 Scaling

Scaling increases or decreases the volume of a solid. The amount of scaling applied with respect to each of the axes are specified using a *scale* vector. If the scaling factors are all zero, scaling is not applied. A scaling transformation maintains the shape and the form of the solid if and only if the scaling factor along all of the axes are equal. This is referred to as *uniform scaling*.

```
147 ⟨Scaling parameters 147⟩ ≡
      struct {
         Vector scale;     /* scaling factor */
      } s;
```

This code is used in chunk 66.

```
148 ⟨Read scaling operation 148⟩ ≡
      ⟨Read scaling parameters 149⟩;
      ⟨Find the target solid for the operation 135⟩;
      ⟨Create scaling parameter node 151⟩;
      ⟨Create and register scaling operator node 152⟩;
```

This code is used in chunk 83.

The scaling parameters are specified in the geometry input file using the following format:

("target" "solid" sx sy sz)

where, "target" is the name of the solid which is obtained after scaling "solid" by the scaling factors sx, sy and sz along the x, y and z axes, respectively. For instance, the scaling specification `("S1" "Solid" 1.0 2.0 3.0)` means, increase the scale of the solid associated with the name "Solid" by a factor of 2.0 and 3.0 respectively along the y and z axes in world coordinate frame and register this intermediate solid as "S1".

```
149 ⟨ Read scaling parameters 149 ⟩ ≡
   read_count = fscanf(f, "(\"%[^\"]\"␣\"%[^\"]\"␣%lf␣%lf␣%lf)", op_target,
         op_solid, &op_x, &op_y, &op_z);
   if (EOF ≡ read_count ∨ 5 ≠ read_count)
      ⟨ Exit after cleanup: failed to read from file 101 ⟩;
```

This code is used in chunk 148.

In order for a scaling to be applicable, the supplied target solid must already exists within the system, either by prior definition as a primitive solid, or as an intermediate solid defined by a CSG subtree.

```
150 ⟨ Find the target solid for the operation 135 ⟩ +≡
   target_solid = find_csg_node(op_solid);
   if (Λ ≡ target_solid) {
      solid_name = op_solid;
      ⟨ Exit after cleanup: solid does not exists 118 ⟩;
   }
```

We now have a valid scaling, so create the parameter leaf node.

```
151 ⟨ Create scaling parameter node 151 ⟩ ≡
   leaf_node = create_csg_node();
   if (Λ ≡ leaf_node) {
      ⟨ Exit after cleanup: failed to create leaf node 137 ⟩;
   }
   else {
      leaf_node→op = PARAMETER;      /* this is a parameter leaf node */
      leaf_node→leaf.s.scale[0] = op_x;
      leaf_node→leaf.s.scale[1] = op_y;
      leaf_node→leaf.s.scale[2] = op_z;
      ⟨ Set up the matrix for scaling 225 ⟩;
   }
```

This code is used in chunk 148.

Finally, create the scaling operator node, and attach the target solid and the parameter node.

```
152 ⟨ Create and register scaling operator node 152 ⟩ ≡
      internal_node = create_csg_node( );
      if (Λ ≡ internal_node) {
         ⟨ Exit after cleanup: failed to create internal node 123 ⟩;
      }
      else {
         internal_node→op = SCALE;      /* this is an operator internal node */
         ⟨ Set line number of node and usage bit of transformed node 139 ⟩;
         ⟨ Set target, parameters and parent pointers 140 ⟩;
         register_csg_node(internal_node, op_target);
         ++(nodes_repo→stat[SCALE]);
      }
```

This code is used in chunk 148.

3.5.9 Registering a solid

A CSG tree defines a solid by specifying how primitive solids are combined using transformation, translation, and boolean operators. The solid thus defined is represented by the root of the CSG tree, and any test against the solid must begin at this root. Hence, if we define n solids in the simulation world, each of these solids must be represented by n CSG trees. We maintain a list of CSG solids using a table of pointers to CSG root nodes.

```
153 ⟨ Read solid registration 153 ⟩ ≡
      ⟨ Read target solid for registration 154 ⟩;
      ⟨ Find the target solid for the operation 135 ⟩;
      ⟨ Register the target solid 155 ⟩;
```

This code is used in chunk 83.

To separate solids from intermediate solid components, we must explicitly register each solid (represented by a node in the single CSG tree) using the following format:

(“target”)

where, “target” is the name of the solid to register.

```
154 ⟨ Read target solid for registration 154 ⟩ ≡
      read_count = fscanf(f, "(\"%[^\"]\")", op_solid);
      if (EOF ≡ read_count ∨ 1 ≠ read_count)
         ⟨ Exit after cleanup: failed to read from file 101 ⟩;
```

This code is used in chunk 153.

155 ⟨Register the target solid 155⟩ ≡
```
process_and_register_solid(target_solid);
set_ubit(target_solid);
```
This code is used in chunk 153.

3.5.10 Defining the simulation world

The simulation world defines the volume of space that we are interested in. Only particles that are inside this volume are tracked during the simulation. Logically, the simulation world is represented by a bounding box, which is axis-aligned with the world coordinate frame.

156 ⟨Global variables 4⟩ +≡
```
BoundingBox sim_world = {ZERO_VECTOR, ZERO_VECTOR};
uint32_t num_subcuboids = 0;
uint32_t div_subcuboids[3] = {1, 1, 1};     /* division along x, y and z */
```

In the geometry input file, the specification of the simulation world uses the following format:

$(lx\ ly\ lz\ ux\ uy\ uz\ l\ m\ n)$

where, the coordinates (lx, ly, lz) and (ux, uy, uz) respectively give the lower and upper bounds. The values l, m and n gives the number of divisions along the x, y and z axes, so that the simulation world is divided into $l \times m \times n$ subcuboids. If there are multiple specifications of the simulation world, the last one read will set the effective bounds.

157 ⟨Read simulation world specification 157⟩ ≡
```
read_count = fscanf(f, "(%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%u␣%u␣%u)",
     &sim_world.l[0], &sim_world.l[1], &sim_world.l[2], &sim_world.u[0],
     &sim_world.u[1], &sim_world.u[2], &div_subcuboids[0], &div_subcuboids[1],
     &div_subcuboids[2]);
if (EOF ≡ read_count ∨ 9 ≠ read_count)
  ⟨Exit after cleanup: failed to read from file 101⟩;
num_subcuboids = div_subcuboids[0] * div_subcuboids[1] * div_subcuboids[2];
if (num_subcuboids > MAX_SUBCUBOIDS)
  ⟨Exit after cleanup: invalid division of simulation world 158⟩;
```
This code is used in chunk 83.

158 ⟨Exit after cleanup: invalid division of simulation world 158⟩ ≡
```
goto invalid_subcuboid_division_exit_after_cleanup;
```
This code is used in chunk 157.

Function *print_sim_world*(*f*) prints the lower and upper bounds of the simulation world to the I/O stream pointed to by *f*.

```
159 ⟨ Global functions 8 ⟩ +≡
  void print_sim_world(FILE *f)
  {
    fprintf(f, "Simulation␣world:␣[%lf,␣%lf,␣%lf␣:␣%lf,␣%lf,␣%lf]\n",
        sim_world.l[0], sim_world.l[1], sim_world.l[2], sim_world.u[0],
        sim_world.u[1], sim_world.u[2]);
  }
```

3.5.11 Bounding Boxes

Every node in the CSG tree is enclosed by a bounding box. This allows us to accelerate the search for solids by efficiently pruning the search trees.

```
160 ⟨ Information common to all CSG nodes 60 ⟩ +≡
  BoundingBox bb;
```

Function *primitive_bb*(*p*, *bb*) calculates the bounding box of a primitive solid *p* and stores the result in the supplied bounding box variable *bb*. This function returns *true* if the bounding box was calculated successfully; otherwise, *false* is returned.

```
161 ⟨ Global functions 8 ⟩ +≡
  bool primitive_bb(Primitive *p, BoundingBox *bb)
  {
    switch (p→type) {
    case BLOCK: ⟨ Calculate bounding box of primitive block 162 ⟩; break;
    case SPHERE: ⟨ Calculate bounding box of primitive sphere 163 ⟩; break;
    case CYLINDER: ⟨ Calculate bounding box of primitive cylinder 164 ⟩; break;
    case TORUS: ⟨ Calculate bounding box of primitive torus 165 ⟩; break;
    default: return false;      /* invalid primitive */
    }
    return true;
  }
```

Note here that the *length*, *height* and *width* are half-lengths of the block's dimension along the x, y and z axes. Furthermore, the block is centered at the origin of the world coordinate frame, so that it's centroid coincides with the origin.

```
162 ⟨ Calculate bounding box of primitive block 162 ⟩ ≡
  bb→l[0] = −p→b.length;
  bb→u[0] = p→b.length;
  bb→l[1] = −p→b.height;
```

$bb \rightarrow u[1] = p \rightarrow b.height;$
$bb \rightarrow l[2] = -p \rightarrow b.width;$
$bb \rightarrow u[2] = p \rightarrow b.width;$

This code is used in chunk 161.

The sphere is centered at the origin of the world coordinate frame.

163 ⟨ Calculate bounding box of primitive sphere 163 ⟩ ≡
$bb \rightarrow l[0] = bb \rightarrow l[1] = bb \rightarrow l[2] = -p \rightarrow s.radius;$
$bb \rightarrow u[0] = bb \rightarrow u[1] = bb \rightarrow u[2] = p \rightarrow s.radius;$

This code is used in chunk 161.

The centroid of the cylinder is centered at the origin of the world coordinate frame, and the normals at the center of the two circular faces of the cylinder are parallel to the y-axis.

164 ⟨ Calculate bounding box of primitive cylinder 164 ⟩ ≡
$bb \rightarrow l[0] = bb \rightarrow l[2] = -p \rightarrow c.radius;$
$bb \rightarrow u[0] = bb \rightarrow u[2] = p \rightarrow c.radius;$
$bb \rightarrow l[1] = -p \rightarrow c.height;$
$bb \rightarrow u[1] = p \rightarrow c.height;$

This code is used in chunk 161.

The torus is centered at the origin so that its center coincides with the origin of the world coordinate frame. Furthermore, the radial surface of the torus lies on the xz-plane.

165 ⟨ Calculate bounding box of primitive torus 165 ⟩ ≡
$bb \rightarrow l[0] = bb \rightarrow l[2] = -(p \rightarrow t.major + p \rightarrow t.minor);$
$bb \rightarrow u[0] = bb \rightarrow u[2] = p \rightarrow t.major + p \rightarrow t.minor;$
$bb \rightarrow l[1] = -p \rightarrow t.minor;$
$bb \rightarrow u[1] = p \rightarrow t.minor;$

This code is used in chunk 161.

Function *intersection_bb*(n, l, r, a) calculates the bounding box n of the intersection node along the supplied axis a using the bounding boxes of the left and right nodes, l and r respectively. The parameter a must only take values 0, 1, and 2, representing respectively the x, y and z axes.

166 **#define** `X_AXIS` 0
#define `Y_AXIS` 1
#define `Z_AXIS` 2

```
⟨ Global functions 8 ⟩ +≡
  bool intersection_bb(BoundingBox *n, BoundingBox *l, BoundingBox
        *r, int a)
  {
    if (l→l[a] < r→l[a] ∧ l→u[a] > r→u[a]) {      /* left fully encloses right */
      n→l[a] = l→l[a];
      n→u[a] = l→u[a];
    }
    else if (r→l[a] < l→l[a] ∧ r→u[a] > l→u[a]) {      /* right fully encloses left */
      n→l[a] = r→l[a];
      n→u[a] = r→u[a];
    }
    else {
      if (l→l[a] < r→l[a]) {      /* intersection with left behind right */
        n→l[a] = r→l[a];
        n→u[a] = l→u[a];
      }
      else {      /* intersection with right behind left */
        n→l[a] = l→l[a];
        n→u[a] = r→u[a];
      }
    }
    return true;
  }
```

Function *apply_affine_matrix*(m, v, r) applies the affine transformation matrix m to the vector v and stores the transformed vector in r. The original vector v is left unmodified. The two macros *affine_normal*( ) and *affine_inverse*( ) use this function to respectively apply the affine matrix for normal transformations, or the inverse transformations (*matrix inverse* of the normal affine matrix).

```
167 #define affine_normal(n, v, r)   apply_affine_matrix((n)→affine, (v), (r))
    #define affine_inverse(n, v, r)   apply_affine_matrix((n)→inverse, (v), (r))
```

```
⟨ Global functions 8 ⟩ +≡
  void apply_affine_matrix(Matrix m, Vector v, Vector r)
  {
    r[0] = m[0][0] * v[0] + m[0][1] * v[1] + m[0][2] * v[2] + m[0][3];
    r[1] = m[1][0] * v[0] + m[1][1] * v[1] + m[1][2] * v[2] + m[1][3];
    r[2] = m[2][0] * v[0] + m[2][1] * v[1] + m[2][2] * v[2] + m[2][3];
    r[3] = 1.0;
  }
```

Function *calculate_bounding_box*(n) calculates the bounding box of the solid represented by the supplied root of a CSG tree. To calculate the bounding box at a given node, the function uses the bounding boxes of the left and right subtrees. Only at the leaves, where we reach a primitive solid, we calculate the bounding box using only the definition of the primitive solid. This function returns *true* if the bounding box was calculated successfully; otherwise, *false* is returned.

```
168 ⟨ Global functions 8 ⟩ +≡
  bool calculate_bounding_box(CSG_Node *n)
  {
    CSG_Node *l, *r;      /* left and right subtrees */
    Vector temp, c[8];      /* for affine transformation */
    int i, j;
    if (is_primitive(n)) {
      if (primitive_bb(n→leaf.p, &n→bb)) {
        ⟨ Calculate affine transformed bounding box 169 ⟩;
        return true;
      }
      else return false;
    }
    if (¬calculate_bounding_box(n→internal.left)) return false;
    if (¬calculate_bounding_box(n→internal.right)) return false;
    ⟨ Find bounding box for the node using left and right subtrees 173 ⟩;
    return true;
  }
```

The bounding box of a primitive solid must be defined using the actual location and orientation of the primitive in the world coordinate frame. This means that, after the bounding box for the primitive has been calculated under the assumption that the center of the primitive coincides with the origin of the world coordinate frame, we must apply the affine transformation for that primitive to the calculated bounding box, to obtain the actual bounds.

```
169 ⟨ Calculate affine transformed bounding box 169 ⟩ ≡
  ⟨ Calculate the eight corners of the bounding box 170 ⟩;
  ⟨ Apply affine transformations to the corners 171 ⟩;
  ⟨ Calculate axis aligned bounding box of the transformed bounding box 172 ⟩;
This code is used in chunk 168.
```

```
170 ⟨ Calculate the eight corners of the bounding box 170 ⟩ ≡
  c[0][0] = c[1][0] = c[2][0] = c[3][0] = n→bb.l[0];
  c[0][1] = c[1][1] = c[4][1] = c[5][1] = n→bb.l[1];
  c[0][2] = c[2][2] = c[4][2] = c[6][2] = n→bb.l[2];
```

```
$c[4][0] = c[5][0] = c[6][0] = c[7][0] = n\rightarrow bb.u[0];$
$c[2][1] = c[3][1] = c[6][1] = c[7][1] = n\rightarrow bb.u[1];$
$c[1][2] = c[3][2] = c[5][2] = c[7][2] = n\rightarrow bb.u[2];$
```
This code is used in chunk 169.

171 ⟨ Apply affine transformations to the corners 171 ⟩ ≡
```
for ($i = 0$; $i < 8$; $++i$) {
    $c[i][3] = 1.0$;      /* homogenise the corner vector */
    $affine\_normal(n, \&c[i][0], temp)$;
    $vector\_copy(\&c[i][0], temp)$;
}
```
This code is used in chunk 169.

After applying an affine transformation, a bounding box may no longer be axis-aligned. Since we require axis-aligned bounding boxes, we must re-calculate an axis-aligned bounding box of the transformed bounding box by using its corners.

172 ⟨ Calculate axis aligned bounding box of the transformed bounding box 172 ⟩ ≡
```
for ($j = 0$; $j < 3$; $++j$) {
    $n\rightarrow bb.l[j] = c[0][j]$;
    $n\rightarrow bb.u[j] = c[0][j]$;
    for ($i = 1$; $i < 8$; $++i$) {
        if ($c[i][j] < n\rightarrow bb.l[j]$) $n\rightarrow bb.l[j] = c[i][j]$;      /* find minimum */
        if ($c[i][j] > n\rightarrow bb.u[j]$) $n\rightarrow bb.u[j] = c[i][j]$;      /* find maximum */
    }
}
for ($i = 0$; $i < 8$; $++i$) $n\rightarrow bb.l[3] = n\rightarrow bb.u[3] = 1.0$;
        /* homogenise bound vectors */
```
This code is used in chunk 169.

173 ⟨ Find bounding box for the node using left and right subtrees 173 ⟩ ≡
```
$l = n\rightarrow internal.left$;
$r = n\rightarrow internal.right$;
switch (BIT_MASK_NODE & $n\rightarrow op$) {
case UNION: ⟨ Calculate bounding box of union 174 ⟩; break;
case INTERSECTION: ⟨ Calculate bounding box of intersection 175 ⟩; break;
case DIFFERENCE: ⟨ Calculate bounding box of difference 176 ⟩; break;
default: return $false$;
}
```
This code is used in chunk 168.

In each of the axes, the lowest of the values stored in the left and right subtree nodes becomes the lower bound for the node. Similarly, the highest value becomes the upper bound. The resulting bounding box must enclose both solids on the left and right subtrees.

```
174 ⟨ Calculate bounding box of union 174 ⟩ ≡
  $n\to bb.l[0] = (l\to bb.l[0] < r\to bb.l[0])$ ? $l\to bb.l[0] : r\to bb.l[0]$;
  $n\to bb.l[1] = (l\to bb.l[1] < r\to bb.l[1])$ ? $l\to bb.l[1] : r\to bb.l[1]$;
  $n\to bb.l[2] = (l\to bb.l[2] < r\to bb.l[2])$ ? $l\to bb.l[2] : r\to bb.l[2]$;
  $n\to bb.u[0] = (l\to bb.u[0] > r\to bb.u[0])$ ? $l\to bb.u[0] : r\to bb.u[0]$;
  $n\to bb.u[1] = (l\to bb.u[1] > r\to bb.u[1])$ ? $l\to bb.u[1] : r\to bb.u[1]$;
  $n\to bb.u[2] = (l\to bb.u[2] > r\to bb.u[2])$ ? $l\to bb.u[2] : r\to bb.u[2]$;
```
This code is used in chunk 173.

When the left and right subtrees do not intersect, a bounding box must not be defined for the node. This is represented simply by making the upper bound smaller than the lower bound—we only need to do this for only one axis; here, we choose the x-axis.

```
175 #define no_intersection_bb(left, right)  ((left).u[0] < (right).l[0] ∨ (left).u[1] <
            (right).l[1] ∨ (left).u[2] < (right).l[2] ∨ (right).u[0] <
            (left).l[0] ∨ (right).u[1] < (left).l[1] ∨ (right).u[2] < (left).l[2])
⟨ Calculate bounding box of intersection 175 ⟩ ≡
  if (no_intersection_bb(l→bb, r→bb)) {
    n→bb.l[0] = 1;
    n→bb.u[0] = −1;
  }
  else {
    intersection_bb(&n→bb, &l→bb, &r→bb, X_AXIS);
    intersection_bb(&n→bb, &l→bb, &r→bb, Y_AXIS);
    intersection_bb(&n→bb, &l→bb, &r→bb, Z_AXIS);
  }
```
This code is used in chunk 173.

For a solid defined by a boolean difference, the bounding box of the node should be the bounding box of the left subtree before we subtract the right subtree.

```
176 ⟨ Calculate bounding box of difference 176 ⟩ ≡
  n→bb = l→bb;
```
This code is used in chunk 173.

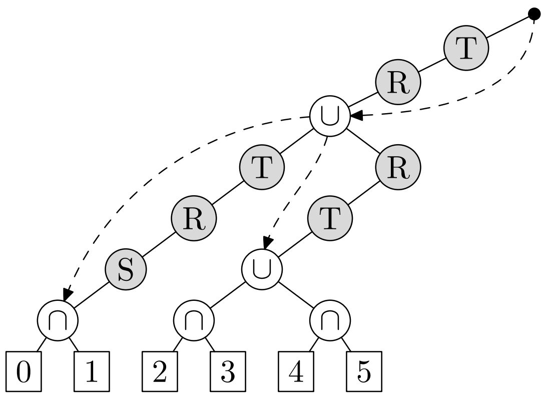
3.5.12 Merge affine transformations

Every node has an affine transformation matrix, which is initialised to the identity matrix. When an affine transformation T is applied to a node, the affine matrix for that node is updated. This affine matrix *affine* gives the accumulated transformations starting at the node to the root of the CSG tree. We also sometimes required the affine matrix *inverse* for applying the transformations in the reverse order, starting at the root until we reach the node.

177 ⟨Information common to all CSG nodes 60⟩ +≡
```
    Matrix affine;      /* inverse affine transformations matrix */
    Matrix inverse;      /* inverse of affine matrix */
```

178 ⟨Initialise affine matrices to the identity matrix 178⟩ ≡
```
    matrix_copy(slot→affine, identity_matrix);
    matrix_copy(slot→inverse, identity_matrix);
```
This code is used in chunk 68.

While the solids are built bottom-up, CSG nodes are added to the nodes repository. When an affine transformation node is added to this repository, the transformation is only applied to one target node, which is an existing node in the nodes repository. However, once the CSG tree has been built, we want each node to store the accumulated affine transformations starting at the root of the CSG tree. Hence, we pre-process the CSG tree so that all of the affine transformations are merged. Furthermore, after the affine transformations have been merged, we do not require the affine transformation nodes and its parameters. Hence, these nodes are then remove from the CSG tree.



The merge begins by first merging transformations starting at the current node. As long as the left-hand child node is an affine transformation node, we accumulate the affine transformation into t, and continue until we have reached a node that is not an affine transformation node. At this node, we store the accumulated affine transformation matrix from t, and detach the list of affine transformation nodes that was merged. Then, we merge the node's subtrees, if they are not leaf nodes. Finally, we the return the last node found so that the parent can update its child pointers after the merge (some nodes may have been removed).

Function *merge_affine*(r) merges the affine transformations on all the paths starting at the root of the CSG tree r and stores the accumulated affine transformation matrix in each of the nodes.

```
180 ⟨ Global functions 8 ⟩ +≡
  CSG_Node *merge_affine(CSG_Node *r)
  {
    CSG_Node *t, *p;      /* temporary node and parent node */
    Matrix mm, m = IDENTITY_MATRIX;      /* accumulated affine matrix */
    if (Λ ≡ r ∨ is_parameter(r)) return r;
    if (r→parent) matrix_copy(m, r→parent→affine);
          /* initialise with parent's affine matrix */
    if (is_translate(r) ∨ is_rotate(r) ∨ is_scale(r)) {
      ⟨ Merge sequence of affine transformation nodes 181 ⟩;
    }
    ⟨ Detach accumulated sequence of affine transformations 182 ⟩;
    ⟨ Merge affine transformation sequences in subtrees 183 ⟩;
    return r;      /* return the first non-affine transformation node */
  }
```

```
181 ⟨ Merge sequence of affine transformation nodes 181 ⟩ ≡
  p = r→parent;      /* parent of the first affine transformation node to merge */
  do {
    t = r;
    matrix_multiply(m, t→internal.right→affine, mm);
        /* accumulate using parameter on the right */
    matrix_copy(m, mm);
    r = t→internal.left;
        /* only left points to the next non-parameter node */
  } while (is_translate(r) ∨ is_rotate(r) ∨ is_scale(r));
  r→parent = p;
      /* update parent for first non-affine node at the end of the sequence */
```
This code is used in chunk 180.

Since affine merge is carried out only once for the entire CSG solid during registration, we do not need to multiply the accumulated affine transformation matrix m with the affine matrix *r→affine* currently available in the node, before replacing the value of *r→affine*. This is because, *r→affine* is always initialised to the `IDENTITY_MATRIX` matrix when CSG nodes are added to the nodes repository.

```
182 ⟨ Detach accumulated sequence of affine transformations 182 ⟩ ≡
  matrix_copy(r→affine, m);
  matrix_inverse(r→affine, m);
  matrix_copy(r→inverse, m);
```
This code is used in chunk 180.

After the affine transformations in the subtrees have been merged, we must update the left and right daughter nodes, which are returned by the recursive subtree merges. It is important to do this because, if the previous daughter nodes were affine transformations, they will now be unavailable after the merge.

```
183 ⟨ Merge affine transformation sequences in subtrees 183 ⟩ ≡
  if (¬is_primitive(r)) {
    if ((t = merge_affine(r→internal.left))) r→internal.left = t;
    if ((t = merge_affine(r→internal.right))) r→internal.right = t;
  }
```
This code is used in chunk 180.

Function *print_csg_tree*(t, l) prints the CSG tree t using inorder tree traversal. The value of l gives the level-of-indentation to use to print the node in the current recursive call.

```
184 ⟨ Global functions 8 ⟩ +≡
  void print_csg_tree(CSG_Node *t, uint32_t l)
  {
    int i;
    if (Λ ≡ t) return;
    if (is_primitive(t)) {
      ⟨ Print primitive solid information 185 ⟩;
      return;
    }
    if (is_parameter(t)) {
      ⟨ Print affine transformation parameters 189 ⟩;
      return;
    }
    print_csg_tree(t→internal.left, l + 1);
    ⟨ Print intermediate node information 190 ⟩;
    print_csg_tree(t→internal.right, l + 1);
  }
```

185 ⟨ Print primitive solid information 185 ⟩ ≡

```
⟨ Print indentation 186 ⟩;
p = t→leaf.p;
switch (p→type) {
case BLOCK:
  printf("BLOCK␣(%u):␣\"%s\"␣%lf␣%lf␣%lf", get_line(t), t→name,
      p→b.length, p→b.width, p→b.height);
  break;
case SPHERE:
  printf("SPHERE␣(%u):␣\"%s\"␣%lf", get_line(t), t→name, p→s.radius);
  break;
case CYLINDER:
  printf("CYLINDER␣(%u):␣\"%s\"␣%lf␣%lf", get_line(t), t→name, p→c.radius,
      p→c.height);
  break;
case TORUS:
  printf("TORUS␣(%u):␣\"%s\"␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf", get_line(t),
      t→name, p→t.phi, p→t.phi_start, p→t.theta, p→t.theta_start, p→t.major,
      p→t.minor);
  break;
default: printf("unknown");
}
⟨ Print bounding box information 187 ⟩;
⟨ Print affine transformation matrices 188 ⟩;
```

This code is used in chunk 184.

186 ⟨ Print indentation 186 ⟩ ≡

```
for (i = 0; i < l; ++i) printf("\t");
```

This code is used in chunks 185, 189, and 190.

187 ⟨ Print bounding box information 187 ⟩ ≡

```
printf("␣[%lf,␣%lf,␣%lf␣:␣%lf,␣%lf,␣%lf]\n", t→bb.l[0], t→bb.l[1], t→bb.l[2],
    t→bb.u[0], t→bb.u[1], t→bb.u[2]);
```

This code is used in chunks 185 and 190.

188 ⟨ Print affine transformation matrices 188 ⟩ ≡

```
matrix_print(stdout, t→affine, 4, 4, l + 1);
printf("\n");
matrix_print(stdout, t→inverse, 4, 4, l + 1);
```

This code is used in chunks 185 and 190.

189 ⟨Print affine transformation parameters 189⟩ ≡
```
⟨Print indentation 186⟩;
switch (BIT_MASK_NODE & t→parent→op) {
case TRANSLATE:
  printf("displacement:␣(%lf,␣%lf,␣%lf)", t→leaf.t.displacement[0],
      t→leaf.t.displacement[1], t→leaf.t.displacement[2]);
  break;
case ROTATE:
  printf("angle:␣%lf,␣axis:␣(%lf,␣%lf,␣%lf)", t→leaf.r.theta,
      t→leaf.r.axis[0], t→leaf.r.axis[1], t→leaf.r.axis[2]);
  break;
case SCALE:
  printf("scaling␣factor:␣(%lf,␣%lf,␣%lf)", t→leaf.s.scale[0],
      t→leaf.s.scale[1], t→leaf.s.scale[2]);
  break;
default: printf("unknown");
}
printf("\n");
```
This code is used in chunk 184.

190 ⟨Print intermediate node information 190⟩ ≡
```
⟨Print indentation 186⟩;
switch (BIT_MASK_NODE & t→op) {
case UNION: printf("UNION␣(%u):␣%s", get_line(t), t→name); break;
case INTERSECTION: printf("INTERSECTION␣(%u):␣%s", get_line(t), t→name);
  break;
case DIFFERENCE: printf("DIFFERENCE␣(%u):␣%s", get_line(t), t→name);
  break;
case TRANSLATE: printf("TRANSLATE␣(%u):␣%s", get_line(t), t→name); break;
case ROTATE: printf("ROTATE␣(%u):␣%s", get_line(t), t→name); break;
case SCALE: printf("SCALE␣(%u):␣%s", get_line(t), t→name); break;
default: printf("unknown");
}
⟨Print bounding box information 187⟩;
⟨Print affine transformation matrices 188⟩;
```
This code is used in chunk 184.

3.5.13 Forest of solids

We record all of the solids by recording the root of their CSG tree. Recording the forest of trees is useful for debugging purposes, and also, while printing out verbose information. Furthermore, generation of the geometry tables are significantly simplified, as we shall see in later sections. We do not use a hash table because all of the solids will be processed sequentially, and there will be no searching.

191
```
#define MAX_CSG_SOLIDS 512
#define reset_forest() memset(&forest_of_solids, 0, sizeof (forest_of_solids))
⟨ Global variables 4 ⟩ +≡
  struct {
    uint32_t n;     /* number of solids in forest */
    CSG_Node *s[MAX_CSG_SOLIDS];     /* root of CSG tree */
  } forest_of_solids;
```

Function *print_forest*( ) prints all of the solids in the forest.

192
```
⟨ Global functions 8 ⟩ +≡
  void print_forest()
  {
    uint32_t i, j;
    for (i = 0; i < forest_of_solids.n; ++i) {
      if (Λ ≡ forest_of_solids.s[i]) continue;
      printf("Solid␣%s:\n\n", forest_of_solids.s[i]→name);
      print_csg_tree(forest_of_solids.s[i], 0);
      printf("\n");
      for (j = 0; j < 80; j++) printf("-");
      printf("\n");
    }
  }
```

Function *process_and_register_solid*(r) registers the solid pointed to by the CSG tree root r. Before registering the solid, the CSG tree is processed to the required form by merging the affine transformations, and by finding the bounding box.

193
```
⟨ Global functions 8 ⟩ +≡
  void process_and_register_solid(CSG_Node *r)
  {
    CSG_Node *t;
    if (Λ ≡ r) return;
    t = merge_affine(r);
        /* returns a non-affine node; otherwise returns Λ */
    if (Λ ≡ t) t = r;     /* in case r was a primitive solid */
    calculate_bounding_box(t);
    forest_of_solids.s[(forest_of_solids.n)++] = t;
  }
```

3.6 Particles inside solids

During the simulation, the *MCS* system must determine which materials a particle is interacting with at each step of its track. Since material properties are associated with solids inside the simulation world, we must first determine the solid which contains the particle at each step of the particle's trajectory. Once we know the solid, we can retrieve the relevant material properties associated with the solid, and then carry out the necessary physics processes.

To find the solid which contains a given particle, we require two algorithms. Firstly, we need an algorithm that will decide if a particle (i.e., point inside the three-dimensional simulation world) is located inside a given solid. This algorithm will traverse the CSG tree that defines the said solid. Secondly, we need an algorithm that will provide us with a list of solids that could potentially contain the particle. In the most basic form, we could do an exhaustive search across all of the solids in the simulation world. However, to improve efficiency, we must use a space-partitioning scheme to prune the search-space.

A containment test must return one of the following:

OUTSIDE	if the point is outside the solid,
INSIDE	if the point is inside the solid,
SURFACE	if point is on the surface, and
INVALID	if either the solid or the vector is undefined.

196 ⟨Type definitions 6⟩ +≡
```
typedef enum {
  OUTSIDE = 0, INSIDE, SURFACE, INVALID
} Containment;
```

3.6.1 Containment inside a solid block

The three pairs of opposite faces of a block determines its containment range on each of the three axes. Hence, we can test if a vector is inside, outside, or on the surface of the block by testing its x, y and z components against the corresponding range.

197 ⟨Information that defines a primitive block 48⟩ +≡
```
double x0, x1, y0, y1, z0, z1;
    /* containment range (i.e., bounding box) */
```

Note here that the dimensions *length*, *height* and *width* correspond respectively to the x, y and z axes in world coordinate frame, and that we are adding or subtracting half-lengths of the respective dimensions.

198 ⟨ Calculate containment range for the block 198 ⟩ ≡
```
  p→b.x0 = −p→b.length;
  p→b.x1 = p→b.length;
  p→b.y0 = −p→b.height;
  p→b.y1 = p→b.height;
  p→b.z0 = −p→b.width;
  p→b.z1 = p→b.width;
```
This code is used in chunk 99.

Let (x, y, z) represent the components of the vector to be tested. Also let the three intervals $[x_0, x_1]$, $[y_0, y_1]$ and $[z_0, z_1]$ respectively define the containment range for the block along the x, y and z axes in the world coordinate frame. Then the point defined by the vector v is:

$$\begin{aligned} \text{outside the block if} &\quad (x < x_0 \vee x > x_1) \vee (y < y_0 \vee y > y_1) \vee (z < z_0 \vee z > z_1), \\ \text{inside the block if} &\quad (x > x_0 \wedge x < x_1) \wedge (y > y_0 \wedge y < y_1) \wedge (z > z_0 \wedge z < z_1), \text{ and} \\ \text{on the surface,} &\quad \text{otherwise.} \end{aligned}$$

In the following implementation, we carry out a surface test instead of the inside test. This avoids boolean conjuctions. Furthermore, we validate the surface test by doing an outside test first.

199 ⟨ Global functions 8 ⟩ +≡
```
  Containment is_inside_block(const Vector v, const Primitive *p)
  {
    if (v[0] < p→b.x0 ∨ v[0] > p→b.x1 ∨ v[1] < p→b.y0 ∨ v[1] > p→b.y1 ∨ v[2] <
          p→b.z0 ∨ v[2] > p→b.z1 ) return OUTSIDE;
    if (v[0] ≡ p→b.x0 ∨ v[0] ≡ p→b.x1 ∨ v[1] ≡ p→b.y0 ∨ v[1] ≡ p→b.y1 ∨ v[2] ≡
          p→b.z0 ∨ v[2] ≡ p→b.z1 ) return SURFACE;
    return INSIDE;
  }
```

3.6.2 Containment inside a solid sphere

We determine the containment of a point inside a sphere by calculating the distance of the point from the sphere's origin. Let δ be the distance of the point from the origin of the sphere. Then the point defined by the vector v is:

$$\begin{aligned} \text{outside the sphere if} &\quad (\delta > radius), \\ \text{inside the sphere if} &\quad (\delta < radius), \text{ and} \\ \text{on the surface,} &\quad \text{otherwise.} \end{aligned}$$

During containment testing, the origin of the sphere always coincides with the origin of the world coordinate frame. Hence, to determine δ, we only need to calculate the magnitude of the vector v.

Note here that we carry out the *outside test* first. This is because, in a real-world setup, a spherical component is less likely to occupy a large volume of the simulation space. Thus, it is highly likely that a particle is outside the sphere most of the time.

```
200 ⟨ Global functions 8 ⟩ +≡
  Containment is_inside_sphere(const Vector v, const Primitive *p)
  {
    double delta = sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
    if (delta > p->s.radius) return OUTSIDE;      /* highly likely */
    if (delta ≡ p->s.radius) return SURFACE;      /* least likely */
    return INSIDE;
  }
```

3.6.3 Containment inside a solid cylinder

During containment testing, the origin of the cylinder coincides with the origin of the world coordinate frame; and the normals at the center of the circular bases of the cylinder are parallel to the y-axis. Hence, to test containment of a point inside a cylinder, we first check if the y-component of vector v is within the range defined by the two parallel circular faces of the cylinder. Secondly, we check if the projected distance of the vector v from the origin on the xz-plane is within the area subscribed on the same plane by the circular surfaces of the cylinder.

```
201 ⟨ Information that defines a primitive cylinder 52 ⟩ +≡
  double y0, y1;      /* containment range of cylinder height */
```

Note here that the dimension *height* corresponds to the y-axis in world coordinate frame, and that we are adding or subtracting half-lengths of the cylinder height.

```
202 ⟨ Calculate containment range for the cylinder 202 ⟩ ≡
  p->c.y0 = −p->c.height;
  p->c.y1 = p->c.height;
This code is used in chunk 107.
```

Let (x, y, z) represent the components of the vector v that we wish to test. Also let δ represent the magnitude of the two-dimensional projection of vector v on the xz-plane, and let the interval $[y_0, y_1]$ give the containment range of the cylinder in the y-axis. Then the point defined by vector v is:

$$\begin{array}{rl} \text{outside the cylinder if} & (y < y_0 \vee y > y_1) \vee (\delta > \textit{radius}), \\ \text{inside the cylinder if} & (y > y_0 \wedge y < y_1) \wedge (\delta < \textit{radius}), \text{ and} \\ \text{on the surface,} & \text{otherwise.} \end{array}$$

Note here that calculation of δ is delayed until it is absolutely necessary.

```
203 ⟨ Global functions 8 ⟩ +≡
  Containment is_inside_cylinder(const Vector v, const Primitive *p)
  {
    double delta;
    if (v[1] < p→c.y0 ∨ v[1] > p→c.y1) return OUTSIDE;
    ⟨ Calculate distance of the two-dimensional xz-projection 204 ⟩;
    if (delta > p→c.radius) return OUTSIDE;
    if (v[1] ≡ p→c.y0 ∨ v[1] ≡ p→c.y1 ∨ delta ≡ p→c.radius) return SURFACE;
    return INSIDE;
  }
```

Since the origin of the cylinder coincides with the origin of the world coordinate frame during containment testing, the magnitude of the two-dimensional projection gives the required distance.

```
204 ⟨ Calculate distance of the two-dimensional xz-projection 204 ⟩ ≡
  delta = sqrt(v[0] * v[0] + v[2] * v[2]);
```
This code is used in chunk 203.

3.6.4 Containment inside a solid torus

During containment testing, the origin of the torus coincides with the origin of the world coordinate frame and is radially symmetrical about the y-axis. Hence, to test containment inside the torus, we first check if the projected distance δ of the vector v on the xz-plane is outside the radial containment range defined by the major and minor radii of the torus. If it is, v is outside.

```
205 ⟨ Information that defines a primitive torus 54 ⟩ +≡
  double r0, r1;   /* radial containment range on xz-plane */
  double phi_end, theta_end;   /* end angles in degrees */
```

```
206 ⟨ Calculate radial containment range for the torus 206 ⟩ ≡
  p→t.r0 = p→t.major − p→t.minor;
  p→t.r1 = p→t.major + p→t.minor;
```
This code is used in chunk 111.

Calculate end angle ϕ_1 after subtending ϕ degrees from the start angle ϕ_0. Do the same for θ.

Note here that $0° < \phi \leq 360°$ and $0° \leq \phi_0 < 360°$. Furthermore, we ensure that $0° < \phi_1 \leq 360°$. This is important while testing if an angle lies *outside* a given range.

```
207 ⟨ Calculate end angles for the torus 207 ⟩ ≡
  p→t.phi_end = p→t.phi_start + p→t.phi;
  p→t.theta_end = p→t.theta_start + p→t.theta;
  if (p→t.phi_end > 360.0) p→t.phi_end −= 360.0;
        /∗ periodicity adjustment ∗/
  if (p→t.theta_end > 360.0) p→t.theta_end −= 360.0;
```
This code is used in chunk 111.

Because of *periodicity*, it is easy to check if an angle $0° \leq \gamma < 360°$ falls outside the region subtended from $0° \leq \phi_0 < 360°$ to $0° < \phi_1 \leq 360°$ by testing one of the following conditions:

$$\text{if } \phi_0 < \phi_1,\ \gamma \text{ is outside if } \gamma < \phi_0 \vee \gamma > \phi_1;$$
$$\text{if } \phi_0 > \phi_1,\ \gamma \text{ is outside if } \gamma < \phi_0 \wedge \gamma > \phi_1.$$

The second condition only occurs when the value of ϕ_1 was adjusted to lie in the range $(0, 360]$. When $\phi_0 = \phi_1$, we could either have a full rotation, or a no rotation. However, since $\phi > 0°$, we must always interpret this as a full rotation; hence, γ is always in range when $\phi_0 = \phi_1$.

```
208 ⟨ Global functions 8 ⟩ +≡
  bool angle_outside_range(double gamma, double phi0, double phi1)
  {
    if (phi0 ≡ phi1) return false;      /∗ full rotation ∗/
    if (phi0 < phi1) return (gamma < phi0 ∨ gamma > phi1);
    if (phi0 > phi1) return (gamma < phi0 ∧ gamma > phi1);
          /∗ periodicity adjusted ∗/
    return false;
  }
```

If v is within the radial containment range, we calculate the angle γ subtended by v on the xz-plane. Let, $\gamma°$ represent the value of γ after converting radians to degrees. If the torus is partial, i.e., $\phi < 360°$, we check if $\gamma°$ is outside the range defined by ϕ_0 and ϕ_1. If $\gamma°$ is not outside the range, we check if v is outside the volume of the tube. Finally, if none of the previous conditions were satisfied, we can be certain that v is either inside the torus, or on the surface. To decide which is valid, we do a final surface test.

Notice that the function *angle_outside_rangle*( ) implicitly tests if the torus is partial (i.e., we do not need to test the condition *p→t.phi* < 360.0 explicitly).

209 ⟨ Global functions 8 ⟩ +≡

```
Containment is_inside_torus(const Vector v, const Primitive *p)
{
  double gamma, gamma_deg, tau, tau_deg, delta, radial;
  Vector tube_center, from_tube_center_to_v, temp;
  ⟨ Calculate the projected distance $\delta$ of $v$ on the $xz$-plane 210 ⟩;
  if (delta < p→t.r0 ∨ delta > p→t.r1) return OUTSIDE;
        /* check radial containment on xz-plane */
  ⟨ Calculate $\gamma$ subtended by $v$ on the $xz$-plane 211 ⟩;
  if (angle_outside_range(gamma_deg, p→t.phi_start, p→t.phi_end))
    return OUTSIDE;
  ⟨ Check if $v$ is outside the tube 212 ⟩;
  ⟨ Check if $v$ is on the surface of the tube 216 ⟩;
  return INSIDE;
}
```

210 ⟨ Calculate the projected distance δ of v on the xz-plane 210 ⟩ ≡

```
delta = sqrt(v[0] * v[0] + v[2] * v[2]);
```

This code is used in chunk 209.

211 ⟨ Calculate γ subtended by v on the xz-plane 211 ⟩ ≡

```
vector_copy(temp, v);
temp[1] = 0.0;      /* angle must be on the xz-plane */
gamma = vector_angle_radian(positive_xaxis_unit_vector, temp);
gamma_deg = convert_radian_to_degree(gamma);
```

This code is used in chunk 209.

The vector *tube_center* gives the center of the tube which subtends γ radians on the xz-plane. Using this point, we calculate *radial*, which is the radial distance of v from *tube_center*. If *radial* is greater than the minor radius, v is outside the torus. If the tube is partial, we must check if the angle τ subtended by the vector *from_tube_center_to_v*, from the *tube_center* to v, is outside θ. If it is, v is outside the volume.

212 ⟨ Check if v is outside the tube 212 ⟩ ≡

```
⟨ Calculate vector tube_center on the center of the tube at $\gamma$ 213 ⟩;
⟨ Calculate radial distance of $v$ from tube_center 214 ⟩;
if (radial > p→t.minor) return OUTSIDE;
⟨ Calculate the radial angle of $v$ on the cross-section at tube_center 215 ⟩;
if (angle_outside_range(tau, p→t.theta_start, p→t.theta_end)) return OUTSIDE;
```

This code is used in chunk 209.

213 ⟨ Calculate vector *tube_center* on the center of the tube at γ 213 ⟩ ≡

```
  $tube\_center[0] = p{\rightarrow}t.major * cos(gamma);$
  $tube\_center[1] = 0.0;$
      /* the circular tube center always lies on the $xz$-plane */
  $tube\_center[2] = p{\rightarrow}t.major * sin(gamma);$
```

This code is used in chunk 212.

214 ⟨ Calculate radial distance of v from *tube_center* 214 ⟩ ≡

```
  $vector\_difference(v, tube\_center, from\_tube\_center\_to\_v);$
  $radial = vector\_magnitude(from\_tube\_center\_to\_v);$
```

This code is used in chunk 212.

215 ⟨ Calculate the radial angle of v on the cross-section at *tube_center* 215 ⟩ ≡

```
  $tau = vector\_angle\_radian(tube\_center, from\_tube\_center\_to\_v);$
  $tau\_deg = convert\_radian\_to\_degree(tau);$
```

This code is used in chunk 212.

216 ⟨ Check if v is on the surface of the tube 216 ⟩ ≡

```
  if ($radial \equiv p{\rightarrow}t.minor$) return SURFACE;
  if ($p{\rightarrow}t.phi < 360.0 \wedge (gamma \equiv p{\rightarrow}t.phi\_start \vee gamma \equiv p{\rightarrow}t.phi\_end)$)
    return SURFACE;
  if ($p{\rightarrow}t.theta < 360.0 \wedge (tau \equiv p{\rightarrow}t.theta\_start \vee tau \equiv p{\rightarrow}t.theta\_end)$)
    return SURFACE;
```

This code is used in chunk 209.

3.6.5 Test containment inside a solid primitive

Containment testing uses different approaches depending on the type of the primitive solid. Nonetheless, during these tests, all of the origins of the primitive solids are assumed to coincide with the origin of the world coordinate frame.

Solid primitives are created internally so that their origin coincides with the origin of the world coordinate frame. Any translation or transformation applied henceforth is then stored as operators in the CSG tree. Thus, while carrying out ⟨ Test containment after affine transformations 221 ⟩, we apply the inverse of the transformations or translations directly to the point being tested, instead of testing the point against the transformed solid. This will become clearer in the following sections.

For now, just remember that all containment tests are carried out assuming that no transformation or translation has been applied to the primitive solid. In other words, the origin of the solid will always coincide with the origin of world coordinate frame, and that the initial orientation of the solid at creation is preserved. Of course, these initial orientations are primitive specific, as discussed in the following sections.

```
217 ⟨ Test containment inside primitive solid 217 ⟩ ≡
  affine_inverse(root, v, r);
  p = root→leaf.p;
  switch (p→type) {
  case BLOCK: return is_inside_block(r, p);
  case SPHERE: return is_inside_sphere(r, p);
  case CYLINDER: return is_inside_cylinder(r, p);
  case TORUS: return is_inside_torus(r, p);
  default: return INVALID;      /* invalid solid */
  }
```
This code is used in chunk 219.

3.6.6 Test containment inside an intermediate solid

For primitive solids, test for containment is pretty straight-forward. We either use distance validation, or parameteric solutions with respect to the solid. However, for intermediate solids, which are defined by a CSG tree, we test containment by traversing the CSG tree. In this section, we use the algorithm described in page 312 of *An Integrated Introduction to Computer Graphics and Geometric Modelling* by Ronald Goldman [CRC Press (2009)]. The function *solid_contains_vector*(*root*, *v*) returns *true* if the vector *v* is inside the solid defined by the CSG tree rooted at *root*; otherwise, *false* is returned. This function assumes that vector *v* is a homogeneous vector.

```
218 ⟨ Global functions 8 ⟩ +≡
  Containment solid_contains_vector(CSG_Node *root, Vector v)
  {
    if (Λ ≡ root ∨ Λ ≡ v) return INVALID;
    return recursively_test_containment(root, v);
  }
```

Function to recursively test containment.

```
219 ⟨ Global functions 8 ⟩ +≡
  Containment recursively_test_containment(CSG_Node *root, Vector v)
  {
    Containment left, right;
    Vector r;      /* used during inverse transformation */
    if (is_primitive(root)) {
      ⟨ Test containment inside primitive solid 217 ⟩;
    }
  else {
    if ($is\_union(root) \lor is\_intersection(root) \lor is\_difference(root)$) {
      ⟨ Test containment in subtrees using boolean operators 220 ⟩;
    }
    else {
      ⟨ Test containment after affine transformations 221 ⟩;
    }
  }
  return INVALID;
}
```

Containment inside boolean solids

We test containment inside intermediate solids (i.e., solids defined as a combination of two solids) using the appropriate boolean tests. When a point is on the surface of either, or both, the left and the right solids, we must check separately if the same point will be inside, or on the surface of the combined solid.

Notice that, when we are carrying out the difference, some points inside the *left* solid will be on the surface of the new solid if they were adjacent to points on the surface of the *right* solid after the subtraction. Since it is difficult to determine this adjacency, we shall assume that points are on the surface of the new solid only if they were on the surface of the left solid.

220 ⟨ Test containment in subtrees using boolean operators 220 ⟩ ≡

```
left = recursively_test_containment(root→internal.left, v);
right = recursively_test_containment(root→internal.right, v);
switch (BIT_MASK_NODE & root→op) {
case UNION:
  if (INVALID ≡ left ∨ INVALID ≡ right) return INVALID;
      /* handle error */
  if (INSIDE ≡ left ∨ INSIDE ≡ right) return INSIDE;
  if (SURFACE ≡ left ∨ SURFACE ≡ right) return SURFACE;
  return OUTSIDE;
case INTERSECTION:
  if (INVALID ≡ left ∨ INVALID ≡ right) return INVALID;
      /* handle error */
  if (OUTSIDE ≡ left ∨ OUTSIDE ≡ right) return OUTSIDE;
  if (INSIDE ≡ left ∧ INSIDE ≡ right) return INSIDE;
  return SURFACE;
case DIFFERENCE:
  if (INVALID ≡ left ∨ INVALID ≡ right) return INVALID;
      /* handle error */
  if (OUTSIDE ≡ right) {
    if (SURFACE ≡ left) return SURFACE;      /* definitely on the surface */
```

```
      if (INSIDE ≡ left) return INSIDE;
              /* could be on the surface, but assume it is inside */
    }
    return OUTSIDE;
  default: return INVALID;
  }
```

This code is used in chunk 219.

Containment after affine transformations

Let T_i represent an *affine transformation* matrix, which expresses one of translation, rotation or scaling, and let T_i^{-1} represent its inverse. Furthermore, let $T = \{T_0, \ldots, T_{n-1}\}$ represent an ordered sequence of n transformations. If s represents a solid, and s' represents the solid after applying T to s, in ascending order starting with T_0, and if r represents the result of applying to a homogeneous vector v the inverse of the transformations in T in descending order starting with T_{n-1}^{-1}, i.e. $s' = T_{n-1} \times T_{n-2} \times \ldots \times T_1 \times T_0 \times s$ and $r = T_0^{-1} \times T_1^{-1} \times \ldots \times T_{n-2}^{-1} \times T_{n-1}^{-1} \times v$, then checking if v lies inside s', is equivalent to checking if r lies inside s. This is because of the matrix property:

$$(T_{n-1} \times T_{n-2} \times \ldots \times T_1 \times T_0)^{-1} = T_0^{-1} \times T_1^{-1} \times \ldots \times T_{n-2}^{-1} \times T_{n-1}^{-1}$$

This significantly simplifies the testing of points using the CSG tree: 1) it is easier to calculate r from v, and 2) containment testing is easier with s than it is with s'.

```
221 ⟨ Test containment after affine transformations 221 ⟩ ≡
  switch (BIT_MASK_NODE & root→op) {
  case TRANSLATE: affine_inverse(root→internal.right, v, r); break;
  case ROTATE: affine_inverse(root→internal.right, v, r); break;
  case SCALE: affine_inverse(root→internal.right, v, r); break;
  default: return INVALID;
  }
  return recursively_test_containment(root→internal.left, r);
```

This code is cited in chunk 217.

This code is used in chunk 219.

The affine matrix T for a translation with displacement vector (x, y, z) is given by:

$$T = \begin{pmatrix} 1.0 & 0.0 & 0.0 & x \\ 0.0 & 1.0 & 0.0 & y \\ 0.0 & 0.0 & 1.0 & z \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

This affine matrix T is stored in the two dimensional array *affine* using *row-major* form.

NOTE: In the following initialisations of the affine matrices, we only set the nonzero elements. All of the node fields have already been initialised to zero during *create_csg_node*( ), since it uses *mem_typed_alloc*( ) which in turn uses the *calloc*( ) system call.

```
222 ⟨ Set up the matrix for translation 222 ⟩ ≡
  leaf_node→affine[0][0] = leaf_node→affine[1][1] = leaf_node→affine[2][2] =
      leaf_node→affine[3][3] = 1.0;
  leaf_node→affine[0][3] = op_x;     /* x-axis translation */
  leaf_node→affine[1][3] = op_y;     /* y-axis translation */
  leaf_node→affine[2][3] = op_z;     /* z-axis translation */
This code is used in chunk 136.
```

The affine matrix R for a rotation of θ radians about the axis specified by a unit vector $u = (x, y, z)$ in the world coordinate frame is given by:

$$R = \begin{pmatrix} tx^2 + c & txy - sz & txz + sy & 0.0 \\ txy + sz & ty^2 + c & tyz - sx & 0.0 \\ txz - sy & tyz + sx & tz^2 + c & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

where, $c = \cos(\theta)$, $s = \sin(\theta)$, and $t = 1 - \cos(\theta)$. This representation is taken from the article *The Mathematics of the 3D Rotation Matrix* by Diana Gruber (http://www.fastgraph.com/makegames/3drotation/).

```
223 ⟨ Local variables: read_geometry( ) 77 ⟩ +≡
  double sine, cosine, t, tx, ty, tz, txy, txz, tyz, sx, sy, sz;
```

The matrix R is stored in the two dimensional array *affine* using *row-major* form.

```
224 ⟨ Set up the matrix for rotation 224 ⟩ ≡
  op_theta *= DEGREE_TO_RADIAN;
  sine = −sin(op_theta);
  cosine = cos(op_theta);
  t = 1.0 − cosine;
  tx = t * op_x;
  ty = t * op_y;
  tz = t * op_z;
  txy = tx * op_y;
  txz = tx * op_z;
  tyz = ty * op_z;
  sx = sine * op_x;
  sy = sine * op_y;
  sz = sine * op_z;
  leaf_node→affine[0][0] = tx * op_x + cosine;
  leaf_node→affine[0][1] = txy − sz;
  leaf_node→affine[0][2] = txz + sy;
  leaf_node→affine[1][0] = txy + sz;
  leaf_node→affine[1][1] = ty * op_y + cosine;
  leaf_node→affine[1][2] = tyz − sx;
  leaf_node→affine[2][0] = txz − sy;
  leaf_node→affine[2][1] = tyz + sx;
  leaf_node→affine[2][2] = tz * op_z + cosine;
  leaf_node→affine[3][3] = 1.0;
```

This code is used in chunk 145.

The affine matrix S for a scaling with scaling factors (x, y, z) is given by:

$$S = \begin{pmatrix} x & 0.0 & 0.0 & 0.0 \\ 0.0 & y & 0.0 & 0.0 \\ 0.0 & 0.0 & z & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

The matrix S is stored in the two dimensional array *affine* using *row-major* form.

225 ⟨ Set up the matrix for scaling 225 ⟩ ≡

```
  leaf_node→affine[0][0] = op_x;     /* x-axis scaling */
  leaf_node→affine[1][1] = op_y;     /* y-axis scaling */
  leaf_node→affine[2][2] = op_z;     /* z-axis scaling */
  leaf_node→affine[3][3] = 1.0;
```

This code is used in chunk 151.

226 ⟨ Test geometry input 226 ⟩ ≡

```
  {
    FILE *f;
    Vector point = ZERO_VECTOR;
    int n;
    Containment flag;
    char c;
    bool t;
    if (false ≡ read_geometry("test/test_geometry_input.data")) exit(1);
    print_geom_statistics(stdout);
    print_sim_world(stdout);
    print_forest();
    if ((f = fopen("test/test_geometry_input_points.data", "r")) ≡ Λ)
      exit(1);
    input_file_current_line = 1;
    while ((c = fgetc(f)) ≠ EOF) {
      ⟨ Discard comments, white spaces and empty lines 79 ⟩;
      ⟨ Process containment test-case 227 ⟩;
    }
  error_invalid_command: fclose(f);
  }
```

227 ⟨ Process containment test-case 227 ⟩ ≡

```
  if (c ≡ 'i' ∨ c ≡ 'o' ∨ c ≡ 's') {
    ⟨ Read parameters for the test-case 228 ⟩;
    ⟨ Validate the test-case 229 ⟩;
  }
  else {
    fprintf(stderr, "Invalid␣test␣command␣'%c'␣at␣line␣%u\n", c,
        input_file_current_line);
    goto error_invalid_command;
  }
```

This code is used in chunk 226.

228 ⟨ Read parameters for the test-case 228 ⟩ ≡

```
  n = fscanf(f, "(\"%[^\"]\"␣%lf␣%lf␣%lf)", op_solid, &point[0], &point[1],
      &point[2]);
  if (EOF ≡ n ∨ 4 ≠ n) {
    printf("Invalid␣%s␣test␣at␣line␣%d\n",
    'i' ≡ c ? "inside" : ('o' ≡ c ? "outside" : "surface"),
        input_file_current_line);
    exit(1);
  }
```

This code is used in chunk 227.

229 ⟨ Validate the test-case 229 ⟩ ≡

```
  temp_node = find_csg_node(op_solid);
  if (Λ ≡ temp_node)
    printf("[%d]␣Solid␣'%s'␣not␣found\n", input_file_current_line, op_solid);
  else {
    vector_homogenise(point);
    flag = solid_contains_vector(temp_node, point);
    t = false;
    switch (flag) {
    case INSIDE: if ('i' ≡ c) t = true; break;
    case SURFACE: if ('s' ≡ c) t = true; break;
    case OUTSIDE: if ('o' ≡ c) t = true; break;
    case INVALID: printf("error:␣"); break;
    }
    printf("[%4d]␣%s␣test:␣%s\n", input_file_current_line,
    'i' ≡ c ? "inside" : ('o' ≡ c ? "outside" : "surface"), t ? "OK" : "Fail");
  }
```

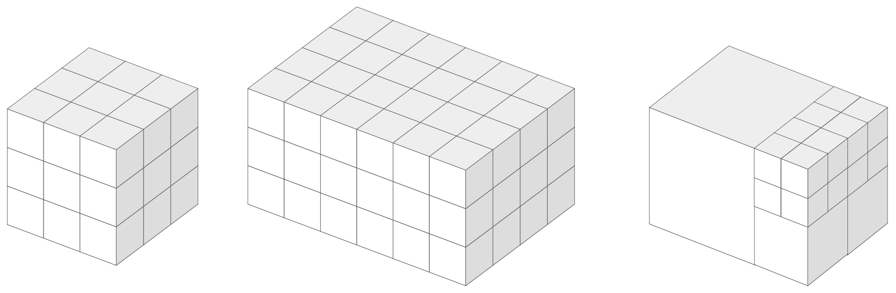
This code is used in chunk 227.

3.7 The simulation world

The simulation world is defined by a *cuboid*, which is composed of solids. Only particles inside this cuboid are tracked during a simulation. To exploit parallelism during simulations, `MCS` logically divides this cuboid into smaller cuboids of equal shape and size, each referred to as a *subcuboid*. A set of subcuboids decomposes the simulation world into disjointed volumes where particles can be tracked independently of other particles in other subcuboids.

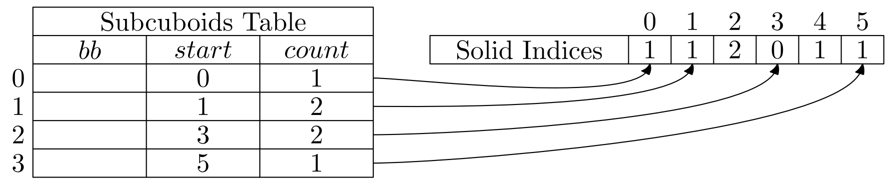
The simulation world must be axis-aligned with the world coordinate frame. It is decomposed into subcuboids $S = \{s_{ijk} : 1 \le i \le l, 1 \le j \le m, 1 \le k \le n\}$, where each of the cuboid's three dimensions has been divided into l, m and n equal parts along the x, y and z axes, respectively. This division can be carried out in any combinations of l, m and n, as long as they are all whole numbers. The other conditions are that all of the subcuboids resulting from a division must all have the same size and shape (i.e., must be a cuboid themselves), must be disjointed, and their union must exactly produce the original cuboid.

For instance, in the following, the two divisions on the left are valid, whereas the right is invalid:



We do not use space-partitioning data structures such as the *octree* or the *kd-tree* to divide the simulation world because it is important that the subcuboids are of the same size and shape. With unequal partitions, we must carry out a lot more calculations in order to determine to which neighbouring subcuboid a particle has escaped. Even worse, this computational overhead is nonuniformly spread differing from one particle to the next, depending on their positions. If we were simulating a few thousand particles, the computational overhead may be considered acceptable. However, since `MCS` is expected to simulate millions, or even billions of particles, the overhead becomes significant and therefore unacceptable. By using subcuboids of the same size and shape, we can avoid all of this overhead by using an efficient lookup-table. We shall discuss this in the following sections. Note, however, that we could still use an octree or a *kd*-tree to accelerate the search for a solid inside a subcuboid.

After the simulation world has been divided, all of the subcuboids are stored in a table, with entries as shown below. We shall refer to this as the *subcuboids table*. The *solid indices buffer* stores indices of solids that are inside a given subcuboid. Hence, for a given subcuboid, the index s gives the starting location of the solids list inside this buffer, and c gives the number of items in that list. For instance, the subcuboid with index 2 contains two solids, whose solid indices are 0 and 1. What these solid indices mean will become clear when we discuss the solids table in later sections.



	Subcuboids Table		
	bb	*start*	*count*
0		0	1
1		1	2
2		3	2
3		5	1

	0	1	2	3	4	5
Solid Indices	1	1	2	0	1	1

231 ⟨ Type definitions 6 ⟩ +≡
```
struct subcuboids_table_item {
  uint32_t s, c;      /* start index and item count in solid indices buffer */
  BoundingBox bb;
     /* enclosing bounding box (coincides with subcuboid boundary) */
};
```

This defines the subcuboids table *ctab* as a component of the geometry table.

232 ⟨ Subcuboids related data structures inside the geometry table 232 ⟩ ≡
```
struct subcuboids_table_item *ctab;
```
See also chunks 235 and 249.

This code is used in chunk 269.

Function *build_subcuboids_table*(bb, l, m, n) builds the subcuboids table by filling in the bounding box information for each of the subcuboids. The dimension of each subcuboid is determined from the size of the original cuboid bb, and the number of equal divisions in each of the dimensions x, y and z as given respectively by l, m and n.

233 ⟨ Global functions 8 ⟩ +≡
```
typedef struct geomtab_struct GeometryTable;
   /* forward-declaration */
bool build_subcuboids_table(GeometryTable *g)
{
  uint32_t i, j, k, c, t;
  double dx, dy, dz, x[2], y[2], z;
  g→ctab = mem_typed_alloc(g→nc, struct subcuboids_table_item,
        mem_phase_two);
  if (Λ ≡ g→ctab) return false;
  t = g→m ∗ g→n;
  dx = (g→sw.u[0] − g→sw.l[0])/g→l;
      /∗ determine subcuboid size using upper and lower bounds ∗/
  dy = (g→sw.u[1] − g→sw.l[1])/g→m;
      /∗ of the cuboid that represents the simulation world ∗/
  dz = (g→sw.u[2] − g→sw.l[2])/g→n;
  x[0] = g→sw.l[0];
  for (i = 0; i < g→l; ++i) {
    x[1] = x[0] + dx;
    y[0] = g→sw.l[1];
    for (j = 0; j < g→m; ++j) {
      y[1] = y[0] + dy;
      z = g→sw.l[2];
      for (k = 0; k < g→n; ++k) {
        c = i ∗ t + j ∗ g→n + k;
        g→ctab[c].bb.l[0] = x[0];
        g→ctab[c].bb.l[1] = y[0];
        g→ctab[c].bb.l[2] = z;
        z += dz;
        g→ctab[c].bb.u[0] = x[1];
        g→ctab[c].bb.u[1] = y[1];
        g→ctab[c].bb.u[2] = z;
      }
      y[0] += dy;
    }
    x[0] += dx;
  }
  return true;
}
```

Function *print_subcuboids_table*(f, g) prints the subcuboids table from the geometry table g to the I/O stream pointed to by f.

234 ⟨ Global functions 8 ⟩ +≡

```
void print_subcuboids_table(FILE ∗f, GeometryTable ∗g)
{
  uint32_t i;
  fprintf(f, "Subcuboids␣Table:\n");
  for (i = 0; i < g→nc; ++i)
    fprintf(f, "%3u␣[%10.3lf,␣%10.3lf,␣%10.3lf␣:␣%10.3l\
        f,␣%10.3lf,␣%10.3lf]\n", i, g→ctab[i].bb.l[0], g→ctab[i].bb.l[1],
        g→ctab[i].bb.l[2], g→ctab[i].bb.u[0], g→ctab[i].bb.u[1], g→ctab[i].bb.u[2]);
}
```

3.7.1 Finding the subcuboid containing a particle

Once the simulation world has been divided into subcuboids, how do we find the subcuboid that contains a given particle? This question arises when we are generating primary particles using a particle gun. The only way to determine the containing subcuboid is to search throughout the simulation world by going through all of the subcuboids. Again, for a few thousand particles the computational cost for an exhaustive search would be acceptable; however, since MCS is expected to simulate millions, we must devise a better strategy.

First of all, we only search for particles that are inside the simulation world. Then, we take advantage of the following four design conditions:

1) all of the subcuboids are of the same size and shape,
2) their union exactly defines the simulation world,
3) no two subcuboids intersect, and
4) the division into subcuboids is immutable throughout the entire simulation.

This means that, we can decompose a single search for the subcuboid in three-dimensional space into three separate, but faster, searches in one-dimensional space. At the end of the three searches, we simply combine the separate results to produce the index of the subcuboid that contains the particle.

235 ⟨ Subcuboids related data structures inside the geometry table 232 ⟩ +≡
  double *∗ctree*;   /∗ stores three binary search trees ∗/

Every node in the binary search tree stores the upper bound of each subcuboid in the selected axis, except for the upper bound that coincides with the upper bound of the simulation world. Hence, if we divide the simulation world into n subcuboids in a given dimension, we require $n - 1$ nodes for the search tree in that dimension. Furthermore, since the division into subcuboids is immutable for the entire simulation, we could improve the search by using a *complete binary tree*, instead of using an arbitrary binary search tree. With a complete binary tree the number of comparisons per search with n subcuboids per dimension is bound by $O(\lceil \lg n \rceil)$. Finally, we can take advantage of the fact that a complete binary search tree can be stored using a one-dimensional array. This not only reduces the space requirements for storing the binary tree, but also simplifies the tree traversal mechanism: Instead of using pointers, we use *bit shift* operators.

Function *build_complete_tree*(t, n, i) recursively builds a complete binary search tree t so that, during an *inorder* traversal, the values in the internal nodes will produce the sequence $s = \{s_i, 0 \leq i < n\}$, where $s_0 = \delta$ and $s_j = s_{j-1} + \delta, 0 < j < n$. The increment δ is the dimension of each subcuboid in the selected axes. The leaf nodes, on the other hand, store partial indices of the subcuboids that contain the value range defined by the internal nodes in the chosen axis. Later on, partial indices from the three axes can be combined to produce a complete subcuboid index.

237 ⟨ Global functions 8 ⟩ +≡

```
static double build_complete_tree_val = 0.0;
    /* upper bound stored in internal node */
static double build_complete_tree_inc = 0.0;    /* size of the subcuboid */
static int build_complete_leaf_idx = 0;
    /* partial subcuboid index stored in leaf */
void build_complete_tree(double *t, unsigned long n, unsigned long i)
{
  unsigned long j;
  if (i < n) {
    j = i << 1;
    build_complete_tree(t, n, j);
    build_complete_tree_val += build_complete_tree_inc;
    t[i] = build_complete_tree_val;
    build_complete_tree(t, n, j + 1);
  }
  else *(unsigned long *) &t[i] = build_complete_leaf_idx++;
}
```

Function *build_subcuboid_binary_search_tree*(t, n, u, l) builds a complete binary search tree t where the upper and lower bounds of the simulation world are respectively u and l units in the selected axis. On the selected axis, the simulation world has been divided into n equal subcuboids.

238 ⟨ Global functions 8 ⟩ +≡

```
void build_subcuboid_binary_search_tree(double *t, unsigned long n, double
        u, double l)
{
  build_complete_tree_val = 0.0;
  build_complete_tree_inc = (u - l)/n;
  build_complete_leaf_idx = 0;
  build_complete_tree(t, n, 1);
}
```

Function *build_subcuboid_search_trees*(g) builds three complete binary search trees using the simulation world data inside the geometry table g. The simulation world is bound by the bounding box bb, and is divided into l, m and n equal parts along the x, y and z axes. There will be a total of $l \times m \times n$ subcuboids, which this function assumes is less than `MAX_SUBCUBOIDS`.

```
239 ⟨ Global functions 8 ⟩ +≡
  bool build_subcuboid_search_trees(GeometryTable *g)
  {
    double *x, *y, *z;
    uint32_t nx, ny, nz;
    ⟨ Allocate memory for the three subcuboid search trees 240 ⟩;
    build_subcuboid_binary_search_tree(x, g→l, g→sw.u[0], g→sw.l[0]);
    build_subcuboid_binary_search_tree(y, g→m, g→sw.u[1], g→sw.l[1]);
    build_subcuboid_binary_search_tree(z, g→n, g→sw.u[2], g→sw.l[2]);
    ⟨ Finalise the subcuboid search trees 241 ⟩;
    return true;
  }
```

This allocates a single array to hold all of the three trees. Each search tree with n subcuboids only requires $n-1$ internal nodes and n leaf nodes. However, we allocate one extra element per tree because the root only begins at the second element (index 1). Instead of wasting this first element, we use it to store the number of nodes. Hence, to store a binary subcuboid search tree with k subcuboids, we allocate an array with $2k$ elements. The same is calculated and allocated for each of the other two axes.

```
240 ⟨ Allocate memory for the three subcuboid search trees 240 ⟩ ≡
  nx = 2 * g→l;
  ny = 2 * g→m;
  nz = 2 * g→n;
  g→ctree = mem_typed_alloc(nx + ny + nz, double, mem_phase_two);
  if (Λ ≡ g→ctree) return false;
  x = g→ctree;
  y = x + nx;
  z = y + ny;
```
This code is used in chunk 239.

Don't forget to store the number of nodes as the first element of the tree array.

```
241 ⟨ Finalise the subcuboid search trees 241 ⟩ ≡
  *(unsigned long *) x = nx;
  *(unsigned long *) y = ny;
  *(unsigned long *) z = nz;
```
This code is used in chunk 239.

Function *print_subcuboid_search_trees*(f, t) prints the three complete binary search trees in t to the I/O stream that is pointed to by f.

242 ⟨ Global functions 8 ⟩ +≡

```
void print_subcuboid_search_trees(FILE *f, double *t)
{
   unsigned long i, j, k, size;
   for (i = 0; i < 3; ++i) {
      size = *(unsigned long *) t;
      k = size/2;
      fprintf(f, "tree␣with␣%lu␣nodes:\n\tinternal:␣", size);
      for (j = 1; j < k; ++j) fprintf(f, "%lf␣", t[j]);
      fprintf(f, "\n\tleaves:␣");
      for ( ; j < size; ++j) fprintf(f, "%lu␣", *(unsigned long *) &t[j]);
      fprintf(f, "\n");
      t += size;      /* move to next tree array */
   }
}
```

Function *find_subcuboid*(t, v) finds the subcuboid that contains a point with position vector v by searching the three binary search trees pointed to by t.

Assume that the simulation world is divided into $l \times m \times n$ subcuboids. After the three one-dimensional searches, let i, j and k represent the search results along the x, y and z axes respectively. The index of the subcuboid is then given by $(i \times m \times n + j \times n + k)$.

243 ⟨ Global functions 8 ⟩ +≡

```
unsigned long find_subcuboid(double *t, Vector v)
{
   unsigned long i, j, c[3], size[3];
   for (i = 0; i < 3; ++i) ⟨ Search for subcuboid in the current tree 244 ⟩;
   return (c[0] * size[1] * size[2] + c[1] * size[2] + c[2]);
}
```

We start at the root and travel left or right, depending on the value at the node, until we reach a leaf, where we have already stored the index of the containing subcuboid.

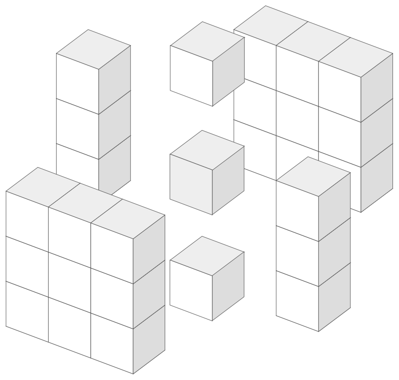
244 ⟨ Search for subcuboid in the current tree 244 ⟩ ≡

```
{
   size[i] = *(unsigned long *) t/2;
   j = 1;
   while (j < size[i]) {
        if $(v[i] < t[j])$ $j \ll= 1$;      /* left subtree */
        else $j = (j \ll 1) + 1$;      /* right subtree */
    }
    $c[i] = *$(unsigned long $*$) $\&t[j]$;      /* index found */
    $t$ += $2 * size[i]$;      /* move to the next binary search tree */
}
```
This code is used in chunk 243.

3.7.2 Relationship between subcuboids

Once inside a simulation, we must determine at each step of the particle's trajectory which subcuboid currently encloses the particle. This is important because in order to apply physics processes to a particle, we must first determine the physical properties of the material with which the particle will interact in that step. To determine the material properties, however, we must know which solid is currently enclosing the particle.

For primary particles that are only starting their trajectory, we must use the function *find_subcuboid*(t, v) to find the containing subcuboid. However, for particles already in simulation, we can devise a better strategy which utilises the current particle position relative to its position in the previous trajectory step. To do this, assume that the planes that divide the simulation world also divide the void outside the simulation world cuboid. Then every subcuboid is surrounded by 26 neighbouring subcuboids, where each neighbour is either part of the simulation world or belongs to the void. This is shown below:



For a given subcuboid, we can define all of its 26 neighbours using the six faces of the subcuboid. Let $f_i, 0 \leq i < 6$ represent the six faces so that the intervals $[f_0, f_1]$, $[f_2, f_3]$, and $[f_4, f_5]$ respectively define the region enclosed by the subcuboid along the x, y and z axes. Now, for every point (x, y, z) in the three-dimensional space, we can define a bit-field $s = \langle s_5, s_4, s_3, s_2, s_1, s_0 \rangle$, where s_0 is the least significant bit. In the following sections, we shall refer to this bit-field as the *s-field*.

All of the bits in s are zero, except in the following cases:
$s_0 = 1$ if $x < f_0$, $s_1 = 1$ if $x > f_1$,
$s_2 = 1$ if $y < f_2$, $s_3 = 1$ if $y > f_3$,
$s_4 = 1$ if $z < f_4$, and $s_4 = 1$ if $z > f_5$.

For instance, points inside the subcuboid will have $s = \langle 000000 \rangle$; whereas, points inside the top neighbour will have $s = \langle 001000 \rangle$. As we can see, the bit pairs (s_0, s_1), (s_2, s_3), and (s_4, s_5) will never be both nonzero. Furthermore, we can represent the faces using a bounding box, where the points (f_0, f_2, f_4) and (f_1, f_3, f_5) are respectively the lower and upper bounds.

Function *update_sfield*(s, bb, v) updates the s-field s for a particle currently at the position vector v using the subcuboid boundary defined by the bounding box bb.

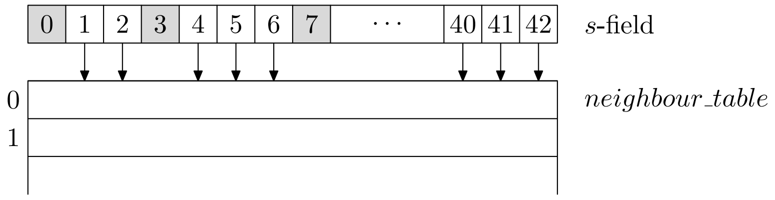
246 ⟨ Global functions 8 ⟩ +≡
```
void update_sfield(uint8_t *s, BoundingBox *bb, Vector v)
{
  *s = #0;
  if (v[0] < bb->l[0]) *s |= #1;      /* s0 */
  else if (v[0] > bb->u[0]) *s |= #2;      /* s1 */
  if (v[1] < bb->l[1]) *s |= #4;      /* s2 */
  else if (v[1] > bb->u[1]) *s |= #8;      /* s3 */
  if (v[2] < bb->l[2]) *s |= #16;      /* s4 */
  else if (v[2] > bb->u[2]) *s |= #32;      /* s5 */
}
```

While a particle is being tracked, we must check if it continues to exists inside the same subcuboid, or if it has escaped to a neighbouring subcuboid. We shall use the s-field to efficiently determine the relevant case for every particle after applying a trajectory step.

Every particle in simulation maintains an s-field, which gives its location relative to the current subcuboid enclosing it. Before applying a trajectory step for the first time, the s-field is initialised once to $\langle 000000 \rangle$. After applying a trajectory step, which could have changed the particle's location, we update the s-field using *update_sfield*(s, bb, v). After subsequent application of a trajectory step, we simply determine the next effective subcuboid using the particle's current s-field, its containing subcuboid, and a neighbourhood table.

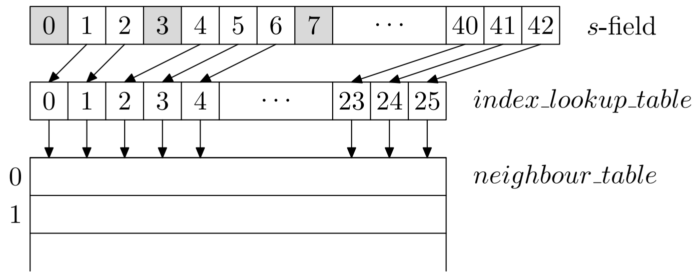
The subcuboid neighbourhood table is an $m \times 26$ two-dimensional array where each row represents one of the m subcuboids, and every element of the row points to one of the 26 neighbouring subcuboids. During lookup, the relevant s-field determines the appropriate column to choose. For instance, if a particle escapes to one of the 26 neighbouring subcuboids, say the top neighbour, it will have the s-field $\langle 001000 \rangle$. To find the subcuboid containing the particle in the next step, we use this as the column during table lookup. So, if the particle is inside the subcuboid with index 10 in the current step, and the s-field is $\langle 001000 \rangle$, the index of the containing subcuboid for the next step is given by the table entry at row 10 and column 8.

If we were to allocate the subcuboid neighbourhood lookup-table using direct-mapping, the table will waste $16 \times$ `MAX_SUBCUBOIDS` table elements. This is because, in order to make every valid s value indexable, we must allocate `MAX_SFIELD` table elements per row. However, out of this row only 26 elements actually contain a valid index to one of the neighbouring subcuboids. This is a consequence of the fact that the bit pairs (s_0, s_1), (s_2, s_3), and (s_4, s_5) will never be both nonzero. Direct mapping, therefore, is space inefficient.



Direct Mapping

If we were to use a two-tiered mapping, however, we only waste space equivalent to 16 table elements. This works by allocating an index lookup-table, which maps the first `MAX_SFIELD` s values to an index that points to one of the 26 valid neighbouring subcuboids.



Tiered Mapping

We now allocate a two-tiered neighbourhood table.

249 **#define** `MAX_SUBCUBOIDS` 1024
#define `NUM_NEIGHBOURS` 26 /* number of cuboid neighbours */
#define `MAX_SFIELD` 42 /* maximum s-field value: $\langle 101010 \rangle$ */

$\langle$ Subcuboids related data structures inside the geometry table 232 $\rangle$ +≡
  int8_t *iltab*[`MAX_SFIELD` + 1]; /* index lookup table */
  uint32_t $**$*ntab*; /* each row containing `NUM_NEIGHBOURS` entries */

Function *get_neighbour*(g, s, i) returns the next effective subcuboid for a particle, where it was previously in subcuboid i and the most recent application of a trajectory step updated the s-field to s. It uses the neighbourhood table in g.

250
```
#define subcuboid_lookup(g, i, s)   ((g)→ntab[(i)][(g)→iltab[s]])
⟨ Global functions 8 ⟩ +≡
  uint32_t get_neighbour(GeometryTable *g, uint32_t i, uint8_t s)
  {
    if (s) return subcuboid_lookup(g, i, s);
    return i;     /* particle continues to exist in the same subcuboid */
  }
```

Function *build_neighbour_table*(g) builds the subcuboid neighbourhood lookup-table inside the geometry table g. The cuboid which represents the simulation world has been divided into l, m and n equal parts along the x, y and z axes. There will be a total of $l \times m \times n$ subcuboids, which this function assumes is less than `MAX_SUBCUBOIDS`.

251
```
#define subcuboid_assign(g, r, s, v)   (g)→ntab[(r)][(g)→iltab[(int)(s)]] = (v)
⟨ Global functions 8 ⟩ +≡
  bool build_neighbour_table(GeometryTable *g)
  {
    uint32_t i, j, k, x, y, z, t = g→m * g→n;
    uint32_t r;     /* row for subcuboid currently being filled in */
    uint8_t s, xb, yb, zb;     /* s-field and extracted bit pairs */
    int8_t q[] = {−1, 0, 1, −1, 2, 3, 4, −1, 5, 6, 7, −1, −1, −1, −1, −1, 8, 9, 10, −1,
          11, 12, 13, −1, 14, 15, 16, −1, −1, −1, −1, −1, 17, 18, 19, −1, 20, 21, 22, −1,
          23, 24, 25};     /* index lookup table data */
    g→ntab = mem_typed_alloc2d(g→nc, NUM_NEIGHBOURS, uint32_t,
          mem_phase_two);
    if (Λ ≡ g→ntab) return false;
    memcpy(g→iltab, q, MAX_SFIELD + 1);
    for (i = 0; i < g→l; ++i)
      for (j = 0; j < g→m; ++j)
        for (k = 0; k < g→n; ++k) {     /* set neighbours */
          r = i * t + j * g→n + k;     /* row index for the current subcuboid */
          for (s = 1; s ≤ MAX_SFIELD; ++s) {
            x = i;
            y = j;
            z = k;
            if (((xb = s & #3) ≡ #3) ∨ ((yb = s & #C) ≡ #C) ∨ ((zb = s & #30) ≡
                  #30)) continue;
            else ⟨ Calculate the neighbour using the axes bit-pairs 252 ⟩;
            }
        }
    return true;
  }
```

252 ⟨ Calculate the neighbour using the axes bit-pairs 252 ⟩ ≡

```
  {
    ⟨ Adjust index of the neighbour cuboid along the x-axis 253 ⟩;
    ⟨ Adjust index of the neighbour cuboid along the y-axis 254 ⟩;
    ⟨ Adjust index of the neighbour cuboid along the z-axis 255 ⟩;
    subcuboid_assign(g, r, s, x * t + y * g→n + z);
  }
```

This code is used in chunk 251.

253 ⟨ Adjust index of the neighbour cuboid along the x-axis 253 ⟩ ≡

```
  if (xb ≡ #1) {
    if (−−x < 0) ⟨ Neighbour subcuboid is outside the simulation world 256 ⟩;
  }
  else if (xb ≡ #2) {
    if (++x ≡ g→l) ⟨ Neighbour subcuboid is outside the simulation world 256 ⟩;
  }
```

This code is used in chunk 252.

254 ⟨ Adjust index of the neighbour cuboid along the y-axis 254 ⟩ ≡

```
  if (yb ≡ #4) {
    if (−−y < 0) ⟨ Neighbour subcuboid is outside the simulation world 256 ⟩;
  }
  else if (yb ≡ #8) {
    if (++y ≡ g→m) ⟨ Neighbour subcuboid is outside the simulation world 256 ⟩;
  }
```

This code is used in chunk 252.

255 ⟨ Adjust index of the neighbour cuboid along the z-axis 255 ⟩ ≡

```
  if (zb ≡ #10) {
    if (−−z < 0) ⟨ Neighbour subcuboid is outside the simulation world 256 ⟩;
  }
  else if (zb ≡ #20) {
    if (++z ≡ g→n) ⟨ Neighbour subcuboid is outside the simulation world 256 ⟩;
  }
```

This code is used in chunk 252.

All of the subcuboids that belong to the void outside the simulation world are given the same subcuboid index `OUTSIDE_WORLD`.

256
```
#define OUTSIDE_WORLD  (MAX_SUBCUBOIDS + 1)
⟨ Neighbour subcuboid is outside the simulation world 256 ⟩ ≡
  {
    subcuboid_assign(g, r, s, OUTSIDE_WORLD);
    continue;
  }
```
This code is used in chunks 253, 254, and 255.

257
```
⟨ Build tables and search trees for managing the subcuboids 257 ⟩ ≡
  build_subcuboid_search_trees(g);
  build_neighbour_table(g);
  build_subcuboids_table(g);
```
This code is used in chunk 272.

Function *print_neighbour_table*(f, g) prints the subcuboid neighbourhood table in g to the I/O stream pointed to by f.

258
```
⟨ Global functions 8 ⟩ +≡
  void print_neighbour_table(FILE *f, GeometryTable *g)
  {
    uint32_t i, j, c, k = MAX_SUBCUBOIDS + 1;
    fprintf(f, "Subcuboid␣neighbourhood␣table:\n");
    for (i = 0; i < g→nc; ++i) {
      for (j = 0; j < NUM_NEIGHBOURS; ++j) {
        c = g→ntab[i][j];
        if (c < k) fprintf(f, "%3d␣", c);
        else fprintf(f, "␣.␣␣");
      }
      fprintf(f, "\n");
    }
  }
```

The usage of the neighbourhood table makes the assumption that particles escaping a subcuboid will be found in one of the 26 surrounding subcuboids. However, this assumption may not apply if the distance that a particle travels after a step application exceeds the dimension of the subcuboid in one of the three axes. In these cases, we must fall-back to finding the subcuboid using *find_subcuboid*(t, v). In most cases, the subcuboid dimensions will be larger than the distance travelled, however, it is important to note the exception. After every step, we must check this distance.

The following code segment tests the functionalities provided by this sections.

```
260 ⟨ Test subcuboid functionalities 260 ⟩ ≡
  {
    uint32_t i, j, k, e, r;
    double dx, dy, dz;
    Vector v;

    do {
      printf("Give␣upper␣bound␣(x,␣y,␣z)␣of␣the␣simulation␣world:\n");
      scanf("%lf␣%lf␣%lf", &geotab.sw.u[0], &geotab.sw.u[1], &geotab.sw.u[2]);
      printf("Give␣lower␣bound␣(x,␣y,␣z)␣of␣the␣simulation␣world:\n");
      scanf("%lf␣%lf␣%lf", &geotab.sw.l[0], &geotab.sw.l[1], &geotab.sw.l[2]);
    } while (geotab.sw.u[0] < geotab.sw.l[0] ∨ geotab.sw.u[1] <
          geotab.sw.l[1] ∨ geotab.sw.u[2] < geotab.sw.l[2]);
    do {
      printf("Give␣number␣of␣divisions␣on␣the␣x,␣y,␣and␣z␣axes:\n");
      scanf("%u␣%u␣%u", &geotab.l, &geotab.m, &geotab.n);
      geotab.nc = geotab.l * geotab.m * geotab.n;
    } while (geotab.nc > MAX_SUBCUBOIDS);
    build_subcuboid_search_trees(&geotab);
    print_subcuboid_search_trees(stdout, geotab.ctree);
    ⟨ Search subcuboid for all points in each of the subcuboids 261 ⟩;
    build_neighbour_table(&geotab);
    build_subcuboids_table(&geotab);
    print_neighbour_table(stdout, &geotab);
    print_subcuboids_table(stdout, &geotab);
  }
```

```
261 ⟨ Search subcuboid for all points in each of the subcuboids 261 ⟩ ≡
  dx = (geotab.sw.u[0] − geotab.sw.l[0])/(double) geotab.l;
  dy = (geotab.sw.u[1] − geotab.sw.l[1])/(double) geotab.m;
  dz = (geotab.sw.u[2] − geotab.sw.l[2])/(double) geotab.n;
  v[0] = geotab.sw.l[0];
  for (i = 0; i < geotab.l; ++i, v[0] += dx) {
    v[1] = geotab.sw.l[1];
    for (j = 0; j < geotab.m; ++j, v[1] += dy) {
      v[2] = geotab.sw.l[2];
      for (k = 0; k < geotab.n; ++k, v[2] += dz) {
        e = i * geotab.m * geotab.n + j * geotab.n + k;
        r = find_subcuboid(geotab.ctree, v);
        if (e ≠ r) goto test_subcuboid_search_failed;
      }
    }
  }
  printf("Test␣was␣a␣success...\n");
  goto test_subcuboid_search_done;
test_subcuboid_search_failed: printf("Failure␣at␣(%u,␣%u,\
      ␣%u):␣%u␣instead␣of␣%u␣for␣(%lf,␣%lf,␣%lf)\n", i, j, k, r, e, v[0],
      v[1], v[2]); test_subcuboid_search_done:
```

This code is used in chunk 260.

Part III

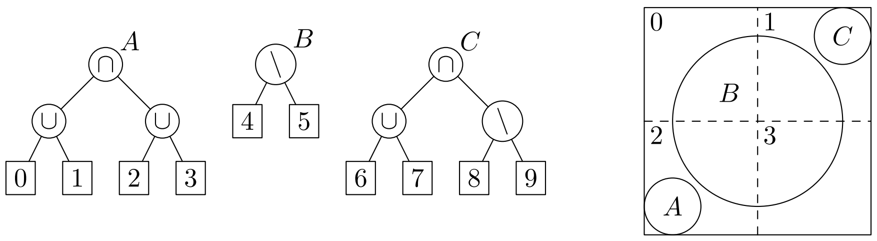
Preparing the shared tables

All of the geometry information must be translated into tables before they can be used during the particle simulations. The aim is to simplify the data structures so that they are efficient, and also allow using devices where management of complex data structures are prohibitive. For instance, GPU memory management relies on the host application; hence, it is complicated to manage data structures that uses pointers.

`MCS` maintains all of the simplified data structures as tables indside a container of tables, known as *geotab*. This is used as a temporary container during the translation and simplification process. Once it is ready, the compact form is transferred to a stage two memory on the devices that will run the simulations.

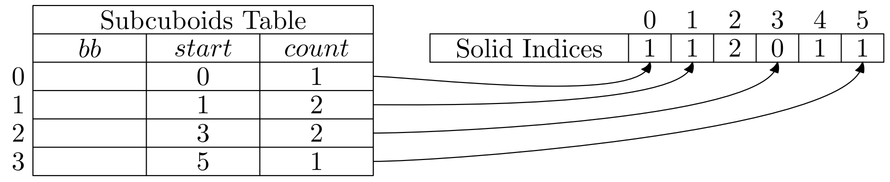
To store information defining the forest of solids and the subcuboids containing them, we could have opted for a straightforward approach where each of the geometry components are stored separately. However, most of the components in *geotab* are related to one another during processing. For instance, searching for the solid which contains a particle requires starting at the simulation world cuboid, and going deeper through the subcuboids, and the CSG tree until we find a containing primitive. Hence, to take advantage of these relationships, the components are stored interlinked using a hierarchcal structure. The tabular representation of this hierarchy is designed so that the cache utilisation is efficient.

Assume that our simulation world is divided into four subcuboids as shown below, and that it contains three solids A, B, and C. The CSG tree for each of the solids are shown on the left-hand side.



According to this diagram, the solid containment scenario is: (0: B), (1: B, C), (2: A, B), (3: B). This is stored in a compact form using the subcuboids table, which we saw in the previous section. This assumes that the solids A, B and C have been assigned the indices 0, 1, and 2 respectively.

We have already seen part of this hierarchical representation when we discussed division of the simulation world into subcuboids, which is reproduced in the following diagram. This captures the link: Simulation world → Subcuboids Table → Solid Indices Buffer.



	Subcuboids Table		
	bb	*start*	*count*
0		0	1
1		1	2
2		3	2
3		5	1

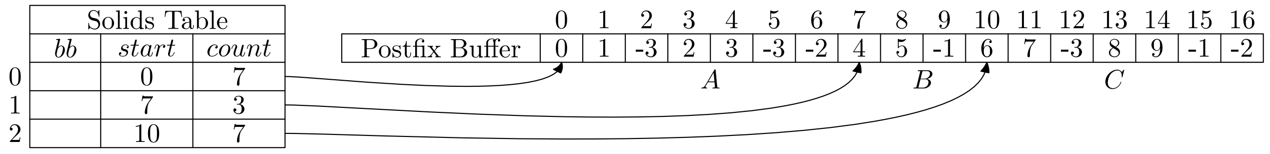
	0	1	2	3	4	5
Solid Indices	1	1	2	0	1	1

This hierarchical linkage can be expanded until we reach the lowest level CSG primitives: Solid Indices → Solids Table → Postfix Expression Buffer → Primitives Table. We shall now discuss this extension.

To simulate a particle, we start by finding the subcuboid that contains the particle. Then, we go to the row in the subcuboids table and retrieve the start index and solids count. Using these values, we retrieve the solids from the solid indices buffer.

For each of the solids indexed by values within the specified range in the buffer, we check if that solid contains the particle. To do this, we then use the *solids table*, which captures information concerning the solids, i.e., the corresponding CSG tree.

A *solids table* is a compact data structure that stores the forest of CSG trees. Instead of storing each of the solids separately, we store the CSG trees as postfix expressions inside a common *postfix expression buffer*. Now, as in the case with the subcuboids table, we store the corresponding start indices and the expression length inside the solids table. This is shown in the following example:

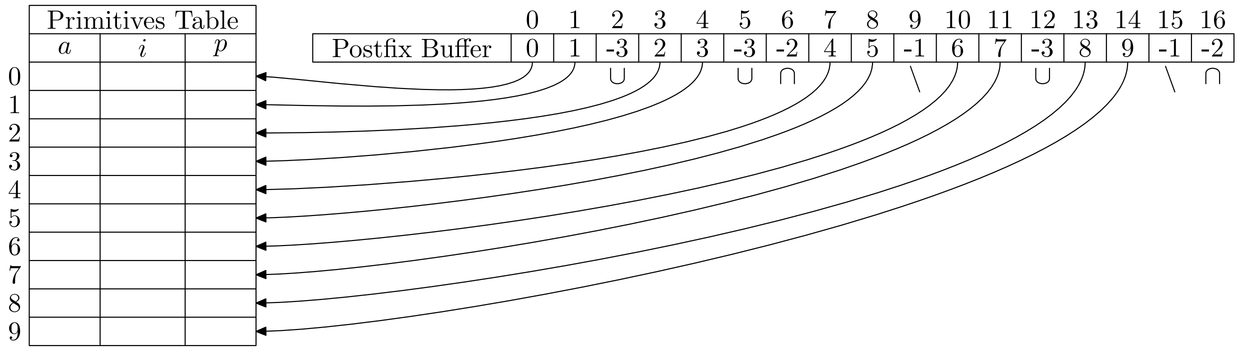


	Solids Table		
	bb	*start*	*count*
0		0	7
1		7	3
2		10	7

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Postfix Buffer	0	1	-3	2	3	-3	-2	4	5	-1	6	7	-3	8	9	-1	-2

266 ⟨ Type definitions 6 ⟩ +≡
```
struct solids_table_item {
  uint32_t s, c;
      /* start index and expression length in postfix expression buffer */
  BoundingBox bb;      /* bounding box for the solid */
};
```

The postfix expression buffer stores indices and operators of a CSG tree. All of the negative integers are operators, where -1 represents a boolean difference, -2 a boolean intersection, and -3 a boolean union. All of the positive integers are indices to the *primitives table*, which store information concerning the parameters specific to the primitive instances.



267 ⟨ Type definitions 6 ⟩ +≡

```
struct primitives_table_item {
   Matrix a, i;      /* accumulated affine and inverse transformations */
   Primitive p;      /* parameters specific to a primitive instance */
};
```

Once we reach the primitives table, we can use the affine transformation matrices (a or i) to transform the particle's position vector, and test containment inside the primitive by using the primitive's parameters available in p. The containment testing inside a primitive is carried out as we evaluate the boolean postfix expression for a given solid.

For instance, if we are testing containment inside the solid B, we will first evaluate containment inside the primitives using rows 4 and 5 in the primitives table, which should return boolean values, and then calculate the boolean difference of the results.

The collection of tables that captures the hierarchical relationship is stored inside a collective data structure, which is defined as the type **GeometryTable**.

269 ⟨ Type definitions 6 ⟩ +≡

```
struct geomtab_struct {
   ⟨ Sizes of the geometry table components 270 ⟩;
   ⟨ Counters used during geometry table generation 271 ⟩;
   ⟨ Subcuboids related data structures inside the geometry table 232 ⟩;
   struct primitives_table_item *p;
   struct solids_table_item *s;
   int32_t *pb;      /* pointer to the postfix expression buffer */
   uint32_t *sb;      /* pointer to the solid indices buffer */
   BoundingBox sw;      /* the simulation world cuboid */
   uint32_t l, m, n;
      /* divisions along x, y and z axes of the simulation world */
} geotab;
```

```
270 ⟨ Sizes of the geometry table components 270 ⟩ ≡
      uint32_t np;    /* number of rows in the primitives table */
      uint32_t ns;    /* number of rows in the solids table */
      uint32_t nc;    /* number of rows in the subcuboids table */
      uint32_t npb;   /* number of entries in the postfix expression buffer */
      uint32_t nsb;   /* number of entries in the solid indices buffer */
```

This code is used in chunk 269.

```
271 ⟨ Counters used during geometry table generation 271 ⟩ ≡
      uint32_t ip;    /* row index within primitives table */
      uint32_t is;    /* row index within solids table */
      uint32_t ic;    /* row index within subcuboids table */
      uint32_t ipb;   /* index within postfix expression buffer */
      uint32_t isb;   /* index within solid indics buffer */
```

This code is used in chunk 269.

Function *create_geotab*(g) generates a data structure g, which consists of all the geometry tables.

```
272 ⟨ Global functions 8 ⟩ +≡
      bool fill_geotab_subcuboids_table(GeometryTable *g);
          /* forward declaration */
      void fill_geotab_csg_table(GeometryTable *g, CSG_Node *n);
      void create_geotab(GeometryTable *g)
      {
        uint32_t i;
        ⟨ Initialise the geometry table 273 ⟩;
        ⟨ Build tables and search trees for managing the subcuboids 257 ⟩;
        ⟨ Fill in the primitives table, the solids table and postfix buffer 274 ⟩;
        ⟨ Check if there are stray node 276 ⟩;
        ⟨ Finalise the geometry table 277 ⟩;
      }
```

To build the geometry table, we must consolidate all of the information concerning the simulation world, subcuboid divisions, solids and their CSG trees, into a coherent and integrated data structure. But before we can do that, we must first initialise the geometry table with the information we collected while processing the user supplied input files. Any inconsistencies (e.g., stray CSG nodes, etc.) will be corrected by later processes.

273 ⟨ Initialise the geometry table 273 ⟩ ≡

```
  g→ip = g→is = g→ic = g→ipb = g→isb = g→nsb = 0;
  g→sw = sim_world;
  g→nc = num_subcuboids;
  g→l = div_subcuboids[0];
  g→m = div_subcuboids[1];
  g→n = div_subcuboids[2];
  g→np = nodes_repo→stat[PRIMITIVE];
  g→npb = nodes_repo→stat[PRIMITIVE] + nodes_repo→stat[UNION] +
      nodes_repo→stat[INTERSECTION] + nodes_repo→stat[DIFFERENCE];
  g→ns = forest_of_solids.n;
  g→p = mem_typed_alloc(g→np, struct primitives_table_item, mem_phase_two);
  g→pb = mem_typed_alloc(g→npb, int32_t, mem_phase_two);
  g→s = mem_typed_alloc(g→ns, struct solids_table_item, mem_phase_two);
```

This code is used in chunk 272.

After processing the geometry specification from the user supplied input file, `MCS` stores all of the solids as a forst of CSG trees, irrespective of the enclosing subcuboids. Hence, we can immediately fill in this information into the respective tables and buffers for each of the solids.

274 ⟨ Fill in the primitives table, the solids table and postfix buffer 274 ⟩ ≡

```
  for (i = 0; i < forest_of_solids.n; ++i) {
    CSG_Node *s = forest_of_solids.s[i];
    if (Λ ≡ s) continue;
    ⟨ Fill table entries for this solid 275 ⟩;
  }
```

This code is used in chunk 272.

275 ⟨ Fill table entries for this solid 275 ⟩ ≡

```
  g→s[g→is].bb = s→bb;
  g→s[g→is].s = g→ipb;      /* current index in the postfix buffer */
  fill_geotab_csg_table(g, s);      /* fill in CSG tree data */
  g→s[g→is].c = g→ipb − g→s[g→is].s;      /* set postfix expression length */
  ++(g→is);      /* moved to next row in solids table */
```

This code is used in chunk 274.

At the beginning, we initialised the geometry table with tentative information collected sequentially while processing the user supplied input files. However, some of this information may be inconsistent with information specified in later segments of the input file. For instance, the user specified stray CSG nodes that are not used in any of the solids. We must correct these errors before finalising the geometry table.

```
276 ⟨ Check if there are stray node 276 ⟩ ≡
  i = g→npb − g→ipb;    /* number of stray nodes */
  if (i) {
    uint32_t j, p = 0;
    fprintf(stderr, "!␣There␣are␣%u␣stray␣nodes␣that␣are␣not\
        ␣in␣any␣of␣the␣solids:\n", i);
    for (j = 0; i ∧ j < MAX_CSG_NODES; ++j) {
      CSG_Node *s = nodes_repo→table[j];
      if (Λ ≡ s ∨ is_inuse(s)) continue;
      if (is_primitive(s)) {
        ++p;
        fprintf(stderr, "\tPrimitive␣\"%s\"␣at␣line␣%u\n", s→name,
            get_line(s));
      }
      else fprintf(stderr, "\tOperator␣\"%s\"␣at␣line␣%u\n", s→name,
            get_line(s));
      −−i;
    }
    g→np −= p;    /* correct the number of active primitives */
    g→npb = g→ipb;    /* correct the number of items in postfix buffer */
  }
```
This code is used in chunk 272.

We are now ready to integrate the solids table with the rest of the simulation world. We will do this by grouping the solids and then assigning them to their enclosing subcuboid.

```
277 ⟨ Finalise the geometry table 277 ⟩ ≡
  fill_geotab_subcuboids_table(g);
```
This code is used in chunk 272.

A solid with a long postfix expression will take longer to evaluate, compared to shorter expressions. Hence, we use *compare_solids*(a, b) to sort the solids table in ascending order using the postfix expression length as the comparison key.

```
278 ⟨ Global functions 8 ⟩ +≡
  static int compare_solids(const void *a, const void *b)
  {
    struct solids_table_item *ap = (struct solids_table_item *) a;
    struct solids_table_item *bp = (struct solids_table_item *) b;
    return (ap→c − bp→c);
  }
```

Until all of the solids have been grouped based on their enclosing subcuboids, we do not know the actual size of the *solid indices buffer*. This is because, a subcuboid can enclose multiple solids, and each of the solids can be enclosed partially by multiple subcuboids. Hence, we cannot build the solid indices buffer immediately in one go; instead, we will build a temporary paged-memory solid indices buffer, and transfer the complete information to the actual geometry table when done.

279 ⟨ Global functions 8 ⟩ +≡

```
bool fill_geotab_subcuboids_table(GeometryTable *g)
{
  Area t, r;     /* memory area for paged solid indices buffer, and a
       temporary pointer */
  uint32_t m = 256;     /* maximum number of items per page */
  uint32_t c = 0;     /* number of items in current page */
  uint32_t *sb;     /* current solid indices buffer page */
  uint32_t i, j;
  mem_init(t);
  mem_init(r);
  ⟨ Fill in the subcuboids table and create a paged solid indices buffer 280 ⟩;
  ⟨ Finalise solid indices buffer by moving paged data to contiguous
       memory 281 ⟩;
  return true;
exit_error: mem_free(t);
  return false;
}
```

We first allocate a solid indices buffer page. Then, we sort the solids in ascending order of the length of their boolean CSG postfix expression. For each of the subcuboids, we then find all of the solids that it encloses. The corresponding indices are then filled into the subcuboids table. When the page is full, we create a new page and continue the process. A subcuboid encloses a solid if their bounding boxes intersect.

280 ⟨ Fill in the subcuboids table and create a paged solid indices buffer 280 ⟩ ≡

```
sb = mem_typed_alloc(m, uint32_t, t);     /* allocate page */
if (Λ ≡ sb) goto exit_error;
qsort(g→s, g→ns, sizeof(struct solids_table_item), compare_solids);
for (i = 0; i < g→nc; ++i) {
  g→ctab[i].s = g→isb;
     /* current index inside the actual solid indices buffer */
  for (j = 0; j < g→ns; ++j) {
    if (no_intersection_bb(g→ctab[i].bb, g→s[j].bb)) continue;
    if (c ≡ m) {
            sb = mem_typed_alloc(m, uint32_t, t);      /* allocate new page */
            if (Λ ≡ sb) goto exit_error;
            c = 0;
        }
        sb[c++] = j;      /* fill in the index inside the solid indices buffer */
        ++g→isb;      /* move to the next available slot */
    }
    g→ctab[i].c = g→isb − g→ctab[i].s;
        /* fill in number of solids enclosed by the subcuboid */
}
```

This code is used in chunk 279.

281 ⟨ Finalise solid indices buffer by moving paged data to contiguous memory 281 ⟩ ≡

```
g→nsb = g→isb;
if (*t) {
    g→sb = mem_typed_alloc(g→nsb, uint32_t, mem_phase_two);
    if (Λ ≡ g→sb) goto exit_error;
    ⟨ Transfer the last page of the solid indices buffer 282 ⟩;
    ⟨ Transfer the remaining pages of the solid indices buffer 283 ⟩;
}
```

This code is used in chunk 279.

Note that the memory pages are maintained inside the memory area as a reverse linked list. Furthermore, we free up the pages after the data have been transferred to the actual solid indices buffer. We must ensure that only the valid contents of the last page is transferred.

282 ⟨ Transfer the last page of the solid indices buffer 282 ⟩ ≡

```
i = g→nsb − c;      /* index in contiguous memory for the last page */
memcpy(&(g→sb[i]), (*t)→first, sizeof(uint32_t) * c);
*r = (*t)→next;
free((*t)→first);
*t = *r;
i −= m;
```

This code is used in chunk 281.

283 ⟨ Transfer the remaining pages of the solid indices buffer 283 ⟩ ≡

```
while (*t) {
    *r = (*t)→next;
    memcpy(&(g→sb[i]), (*t)→first, sizeof(uint32_t) * m);
    free((*t)→first);
    *t = *r;
    i −= m;
  }
```

This code is used in chunk 281.

Function *fill_geotab_csg_table*(g, n) fill in the CSG tree information for a solid with CSG root s inside the geometry table g. It uses recursive post-order tree traversal.

284 ⟨ Global functions 8 ⟩ +≡

```
  void fill_geotab_csg_table(GeometryTable *g, CSG_Node *n)
  {
    if (Λ ≡ g ∨ Λ ≡ n) return;
    if (is_primitive(n)) {
      matrix_copy(g→p[g→ip].a, n→affine);
      matrix_copy(g→p[g→ip].i, n→inverse);
      g→p[g→ip].p = *(n→leaf.p);
      g→pb[(g→ipb)++] = (g→ip)++;
      return;
    }
    fill_geotab_csg_table(g, n→internal.left);
    fill_geotab_csg_table(g, n→internal.right);
    switch (BIT_MASK_NODE & n→op) {
    case UNION: g→pb[g→ipb++] = BOOLEAN_UNION; break;
    case INTERSECTION: g→pb[g→ipb++] = BOOLEAN_INTERSECTION; break;
    case DIFFERENCE: g→pb[g→ipb++] = BOOLEAN_DIFFERENCE; break;
    default: ;
    }
  }
```

285 ⟨ Global functions 8 ⟩ +≡

```
  void print_geotab(FILE *f, GeometryTable *g)
  {
    uint32_t i, j;
    fprintf(f, "G␣E␣O␣M␣E␣T␣R␣Y␣␣T␣A␣B␣L␣E\nPrimitives␣table:\n");
    for (i = 0; i < g→np; ++i) {
      fprintf(f, "Affine:\n");
      matrix_print(f, g→p[i].a, 4, 4, 0);
      fprintf(f, "\nInverse:\n");
      matrix_print(f, g→p[i].i, 4, 4, 0);
      for (j = 0; j < 80; ++j) fprintf(f, "-");
      fprintf(f, "\n");
    }
    fprintf(f, "Solids␣table:\n");
    for (i = 0; i < g→ns; ++i) fprintf(f, "%u␣%u\n", g→s[i].s, g→s[i].c);
    fprintf(f, "Postfix␣buffer:\n");
    for (i = 0; i < g→npb; ++i) fprintf(f, "%d␣", g→pb[i]);
    fprintf(f, "\nSubcuboids␣table:\n");
    for (i = 0; i < g→nc; ++i) fprintf(f, "%u␣%u\n", g→ctab[i].s, g→ctab[i].c);
    fprintf(f, "Solid␣indices␣buffer:\n");
    for (i = 0; i < g→nsb; ++i) fprintf(f, "%u␣", g→sb[i]);
    fprintf(f, "\n");
    print_subcuboid_search_trees(stdout, geotab.ctree);
    print_neighbour_table(stdout, &geotab);
    print_subcuboids_table(stdout, &geotab);
  }
```

286 ⟨ Test geometry table generation 286 ⟩ ≡

```
  {
    if (false ≡ read_geometry("test/test_gpu_table.data")) exit(1);
    print_geom_statistics(stdout);
    create_geotab(&geotab);
    print_geotab(stdout, &geotab);
  }
```

3.8 Boolean stack

We use a boolean stack to evaluate the CSG boolean expression.

287 **#define** `MAX_BOOLEAN_STACK_SIZE` 1024

```
⟨ Type definitions 6 ⟩ +≡
  typedef struct {
    int tos, size;
    bool v[MAX_BOOLEAN_STACK_SIZE];
  } boolean_stack;
```

288
```
⟨ Global functions 8 ⟩ +≡
  bool boolean_stack_init(boolean_stack *s)
  {
    if (Λ ≡ s) return false;
    s→tos = 0;
    s→size = MAX_BOOLEAN_STACK_SIZE;
    return true;
  }
```

289
```
⟨ Global functions 8 ⟩ +≡
  bool boolean_stack_push(boolean_stack *s, bool v)
  {
    if (s→tos ≡ s→size) return false;
    s→v[s→tos++] = v;
    return true;
  }
```

290
```
⟨ Global functions 8 ⟩ +≡
  bool boolean_stack_pop(boolean_stack *s, bool *v)
  {
    if (0 ≡ s→tos) return false;
    *v = s→v[−−s→tos];
    return true;
  }
```

3.9 Evaluation of CSG boolean expression

This section is concerned with checking if a three-dimensional point lies inside a solid, where the solid is represented by a postfix boolean expression.

Function *is_inside_primitive*(v, p) checks if the vector v lies inside the primitive p. It returns *true* if v is inside p; or *false*, otherwise. The value of v and p are both left unmodified.

292 ⟨ Global functions 8 ⟩ +≡
```
bool is_inside_primitive(const Vector v, const Primitive *p)
{
  Containment c;
  switch (p→type) {
  case BLOCK: c = is_inside_block(v, p); break;
  case SPHERE: c = is_inside_sphere(v, p); break;
  case CYLINDER: c = is_inside_cylinder(v, p); break;
  case TORUS: c = is_inside_torus(v, p); break;
  default: return false;      /* invalid solid */
  }
  if (INSIDE ≡ c ∨ SURFACE ≡ c) return true;
  return false;
}
```

Function *is_inside*(v, s, r) checks if the vector v is inside the solid that is indexed by the value stored at index s inside the *solid indices buffer*. To carry out the containment test, the boolean postfix expression that corresponds to the CSG tree of the solid is evaluated against the vector v. If this evaluation was successful, the function returns *true*, and the actual containment result is stored in r. If there was an error, however, e.g., the stack was full, the function will return a *false*. In this case, value in r must be discarded.

NOTE: The performance of this evaluator can be improved significantly by first normalising the CSG tree into sum-of-product form. We have not pursue this optimisation here due to time constraints. Normalisation to sum-of-product form can be done using the algorithm described by Jack Goldfeather, Steven Molnar, Greg Turk, and Henry Fuchs in *Near Real-Time CSG Rendering Using Tree Normalization and Geometric Pruning* [IEEE Computer Graphics and Applications, pp. 20–28, May **1989**].

293
```
#define BOOLEAN_DIFFERENCE −1
#define BOOLEAN_INTERSECTION −2
#define BOOLEAN_UNION −3
#define BOOLEAN_INVALID −4
```

```
⟨ Global functions 8 ⟩ +≡
  bool is_inside(Vector v, uint32_t solid, bool *result)
  {
    uint32_t start, end;      /* start and end indices */
    int item = BOOLEAN_INVALID;      /* current postfix item */
    boolean_stack stack;
    boolean_stack_init(&stack);
    ⟨ Retrieve start index and length of the postfix expression 294 ⟩;
    ⟨ Evaluate the boolean postfix expression using the stack 295 ⟩;
    ⟨ Check if the CSG expression was valid and retrieve the result 298 ⟩;
    return true;
    ⟨ Handle irrecoverable error: is_inside(v, s, result) 299 ⟩;
    return false;
  }
```

We use the *solids table* component of the geometry table to retrieve the start index and length of the boolean postfix expression representing the solid. These values will then be used to retrieve the boolean expression from the *postfix expression buffer*.

```
294 ⟨ Retrieve start index and length of the postfix expression 294 ⟩ ≡
      start = geotab.s[solid].s;
      end = start + geotab.s[solid].c;
```

This code is used in chunk 293.

The CSG expression is an array of integers, where all of the positive values give an index inside the *primitives table* component of the geomatry table. Any negative value must be either -1, -2, or -3, denoting respectively a boolean difference, intersection, or union. Any other value in the expression is invalid. For instance, the integer sequence $c = \{0, 3, -2, 1, -1, 2, 3, -2, -3\}$ is a postfix representation for the boolean expression $((A \cap D) - B) \cup (C \cap D)$, where the primitives array $p = \{A, B, C, D\}$.

```
295 ⟨ Evaluate the boolean postfix expression using the stack 295 ⟩ ≡
      while (start < end) {
        item = geotab.pb[start];      /* lookup current item */
        if (BOOLEAN_DIFFERENCE < item) {
          ⟨ Push to stack the boolean containment of v inside primitive 296 ⟩;
        }
        else {
          ⟨ Evaluate boolean operator and push result into stack 297 ⟩;
        }
        ++start;
}
```
This code is used in chunk 293.

We lookup the *primitives table* component of the geometry table to retrieve the primitive p that corresponds to the current postfix item. We then call *is_inside_primitive*(v, p) to check if the vector v is inside the primitive p. The returned value is then pushed to the stack, to be retrieved later as an operand to a subsequent boolean operator.

296 ⟨ Push to stack the boolean containment of v inside primitive 296 ⟩ ≡
```
bool t = is_inside_primitive(v, &geotab.p[item].p);
if (¬boolean_stack_push(&stack, t)) goto stack_full;
```
This code is used in chunk 295.

The first stack pop gives the right operand r, and the second gives the left operand l. Since, boolean difference is noncommutative, we must preserve the order during its evaluation.

297 ⟨ Evaluate boolean operator and push result into stack 297 ⟩ ≡
```
bool l, r;     /* left and right operands */
if (¬boolean_stack_pop(&stack, &r)) goto stack_empty;
if (¬boolean_stack_pop(&stack, &l)) goto stack_empty;
switch (item) {
case BOOLEAN_DIFFERENCE: if (¬boolean_stack_push(&stack, l ∧ ¬r))
    goto stack_full; break;
case BOOLEAN_INTERSECTION: if (¬boolean_stack_push(&stack, l ∧ r))
    goto stack_full; break;
case BOOLEAN_UNION: if (¬boolean_stack_push(&stack, l ∨ r)) goto stack_full;
    break;
default: fprintf(stderr, "Invalid␣value␣'%d'␣in␣expression\n", item);
  return false;
}
```
This code is used in chunk 295.

A CSG boolean expression is valid if the stack is empty after the result has been popped out. Hence, the following test pop for t must fail for valid CSG expressions.

298 ⟨ Check if the CSG expression was valid and retrieve the result 298 ⟩ ≡
```
{
  bool t;
```

```
    boolean_stack_pop(&stack, result);      /* must be the only item in stack */
    if (boolean_stack_pop(&stack, &t)) {
      fprintf(stderr, "Invalid␣CSG␣tree␣expression\n");
      return false;
    }
  }
```

This code is used in chunk 293.

299 ⟨ Handle irrecoverable error: *is_inside*(*v*, *s*, *result*) 299 ⟩ ≡

```
stack_empty: fprintf(stderr, "Boolean␣stack␣is␣empty...␣");
  goto exit_error;
stack_full: fprintf(stderr, "Boolean␣stack␣is␣full...␣"); goto exit_error;
exit_error:
  fprintf(stderr, "while␣evaluating␣'%d'␣at␣index␣%d\n", item, start);
```

This code is used in chunk 293.

Function *solid_contains_particle*(*s*, *p*) checks if the solid *s* contains the particle *p*. If it does, *true* is returned; otherwise, *false* is returned. To carry out the actual check, this function tests the containment of the particle's position vector *p*→*v* against the CSG expression that defines the solid.

300 ⟨ Global functions 8 ⟩ +≡

```
  bool solid_contains_particle(uint32_t s, Particle *p)
  {
    bool result;
    if (is_inside(p→v, s, &result)) return result;
    else return false;
  }
```

Part IV

GPU Memory Management using Nvidia CUDA

To use CUDA threads, we must make all of the relevant data structures available on the device memory. Hence, similar to the host CPU, we must implement a strategy for memory allocation and deallocation on the GPU devices. We use memory areas, but with CUDA APIs for memory allocation. Note that we use *cudaMalloc*( ), hence, we must not forget to initialise the memory using *cudaMemset*( ), since there is no interface similar to *calloc*( ).

301 ⟨ Type definitions 6 ⟩ +≡

```
  typedef struct memory_area *GPUArea[1];
```

302#define *gpu_memblocks_per_hostblock* 32 /* 512 bytes per block */

⟨ Global functions 8 ⟩ +≡

```
  static void **next_gpu_memblock = Λ;
  static void **bad_gpu_memblock = Λ;
  void **create_gpu_memblock(GPUArea s)
  {
    void **slot = next_gpu_memblock;
    if (slot ≡ bad_gpu_memblock) {
      slot = mem_typed_alloc(gpu_memblocks_per_hostblock, void *, s);
      if (Λ ≡ slot) return Λ;
      else {
        next_gpu_memblock = slot + 1;
        bad_gpu_memblock = slot + gpu_memblocks_per_hostblock;
      }
    }
    else next_gpu_memblock++;
    return slot;
  }
```

Function *cuda_mem_free*(*s*) frees the memory blocks in the GPU device currently allocated under the memory area *s*. We start at the head of the linked list and free each of the blocks. At the end of this function, the memory area variable *s* will be in a state similar to the one when it was first initialised.

303 ⟨ Global functions 8 ⟩ +≡

```
  void cuda_mem_free(GPUArea s)
  {
    Area t;
    uint16_t i;
    while (*s) {
      *t = (*s)→next;
      for (i = 0; i < gpu_memblocks_per_hostblock ∧ (*s)→first + i; ++i)
        cudaFree(*(void **)((*s)→first + i));
      free((*s)→first);
      *s = *t;
    }
  }
```

To allocate storage space to fit n elements of a given data type t inside the GPU device under the memory area s, we use the C/ macro *cuda_mem_typed_alloc*(n, t, s).

304 **#define** *cuda_mem_typed_alloc*(n, t, s)

```
          ( t * ) cuda_mem_alloc((n) * sizeof (t), s)
```

Function *cuda_mem_alloc*(n, s) allocates a block of storage space inside the GPU device under the memory area s that will fit n consecutive bytes of data. In addition to the raw data, a block must also allocate enough space to store the two pointers defined by a memory area. These are used to implement the linked list of memory blocks. The memory area pointers are stored at the end of each block. To improve efficiency of memory allocation, we allocate a block as a consecutive array of 256 bytes. The details of the implementation are similar to that of *mem_alloc*(n, s), so we will not repeat them here.

305 ⟨ Global functions 8 ⟩ +≡

```
  void *cuda_mem_alloc(size_t n, GPUArea s)
  {
    size_t m = sizeof(void *), q;      /* size of a pointer variable */
    void *block, **t;      /* address of the new block */
    if (1 > n ∨ (#ffff00 − 2 * m) < n) return Λ;
    n = round_up_mult(n, m);      /* round up n to a multiple of m */
    q = (size_t)((n + 255)/256) * 256;
    if (cudaSuccess ≡ cudaMalloc((void **) &block, q)) {
      t = create_gpu_memblock(s);
      if (Λ ≡ t) {
        cudaFree(block);
        return Λ;
      }
      cudaMemset(block, 0, q);      /* get the calloc( ) effect */
      *t = block;
    }
    return block;
  }
```

To allocate a two-dimensional array inside the GPU device that will fit $r \times c$ elements of a given data type t under the memory area s, we use the C/ macro *cuda_mem_typed_alloc2d*(p, r, c, t, s). The pitch value is returned in p, which is the width in bytes of the two-dimensional allocation. It is used to calculate the two-dimensional address of the array elements.

306 **#define** *cuda_mem_typed_alloc2d*(p, r, c, t, s)
```
        ( t * ) cuda_mem_alloc_2d((p), (r), (c), sizeof (t), s)
```

307 ⟨ Global functions 8 ⟩ +≡
```
  void *cuda_mem_alloc_2d(size_t *p, uint32_t r, uint32_t c, size_t
        s, GPUArea a)
  {
    void *block, **t;
    if (cudaSuccess ≡ cudaMallocPitch((void **) &block, p, c * s, r)) {
      t = create_gpu_memblock(a);
      if (Λ ≡ t) {
        cudaFree(block);
        return Λ;
      }
      cudaMemset2D(block, *p, 0, c * s, r);     /* get the calloc( ) effect */
      *t = block;
    }
    return block;
  }
```

We shall use only one GPU memory area for the entire application.

308 ⟨ Global variables 4 ⟩ +≡
```
  GPUArea mem_gpu = {Λ};     /* GPU memory area */
```

3.10 Memory organisation

Inside the GPU memory, we create a table which contains the geometry information and the necessary physics tables. This has the following structure:

309 ⟨ Type definitions 6 ⟩ +≡
```
typedef struct gpu_table {
  uint32_t nc;   /* number of entries in subcuboids table */
  uint32_t ns;   /* number of entries in solids table */
  uint32_t np;   /* number of entries in primitives table */
  uint32_t npb;   /* number of entries in postfix expression buffer */
  uint32_t nsb;   /* number of entries in solid indices buffer */
  uint32_t l, m, n;   /* divisions along x, y and z axes */
  uint32_t nct;   /* number of items in the subcuboids search tree */
  struct subcuboids_table_item *ctab;   /* subcuboids lookup table */
  struct solids_table_item *s;   /* solids lookup table */
  struct primitives_table_item *p;   /* primitives lookup table */
  int32_t *pb;   /* pointer to the postfix expression buffer */
  uint32_t *sb;   /* pointer to the solid indices buffer */
  double *ctree;   /* subcuboid search tree */
  int8_t *iltab;   /* index lookup table for two-tiered neighbour table */
  uint32_t *ntab;   /* subcuboid neighbour table */
  size_t pitch;   /* width in bytes for the neighbour table */
  BoundingBox sw;   /* the simulation world cuboid */
} GPUTables;
```

310 ⟨ Global functions 8 ⟩ +≡
```
GPUTables *transfer_tables_to_gpu(GeometryTable *g)
{
  GPUTables *t, k;
  k.nc = g→nc;
  k.ns = g→ns;
  k.np = g→np;
  k.npb = g→npb;
  k.nsb = g→nsb;
  k.l = g→l;
  k.m = g→m;
  k.n = g→n;
  k.sw = g→sw;
  k.ctab = cuda_mem_typed_alloc(k.nc, struct subcuboids_table_item,
      mem_gpu);
  k.s = cuda_mem_typed_alloc(k.ns, struct solids_table_item, mem_gpu);
```

```
    k.p = cuda_mem_typed_alloc(k.np, struct primitives_table_item,
        mem_gpu);
    k.pb = cuda_mem_typed_alloc(k.npb, int32_t, mem_gpu);
    k.sb = cuda_mem_typed_alloc(k.nsb, uint32_t, mem_gpu);
    k.ctree = cuda_mem_typed_alloc(2 * (k.l + k.m + k.n), double, mem_gpu);
    k.iltab = cuda_mem_typed_alloc(MAX_SFIELD + 1, int8_t, mem_gpu);
    k.ntab = cuda_mem_typed_alloc2d(&k.pitch, k.nc, NUM_NEIGHBOURS,
        uint32_t, mem_gpu);
    t = cuda_mem_typed_alloc(1, GPUTables, mem_gpu);
    cudaMemcpy(k.ctab, g→ctab, k.nc * sizeof(struct subcuboids_table_item),
        cudaMemcpyHostToDevice);
    cudaMemcpy(k.s, g→s, k.ns * sizeof(struct solids_table_item),
        cudaMemcpyHostToDevice);
    cudaMemcpy(k.p, g→p, k.np * sizeof(struct primitives_table_item),
        cudaMemcpyHostToDevice);
    cudaMemcpy(k.pb, g→pb, k.npb * sizeof(int32_t),
        cudaMemcpyHostToDevice);
    cudaMemcpy(k.sb, g→sb, k.nsb * sizeof(uint32_t),
        cudaMemcpyHostToDevice);
    k.nct = 2 * (k.l + k.m + k.n);
    cudaMemcpy(k.ctree, g→ctree, k.nct * sizeof(double),
        cudaMemcpyHostToDevice);
    cudaMemcpy(k.iltab, g→iltab, (MAX_SFIELD + 1) * sizeof(int8_t),
        cudaMemcpyHostToDevice);
    cudaMemcpy2D(k.ntab, k.pitch, g→ntab, NUM_NEIGHBOURS * sizeof(uint32_t),
        NUM_NEIGHBOURS * sizeof(uint32_t), k.nc, cudaMemcpyHostToDevice);
    cudaMemcpy(t, &k, sizeof (k), cudaMemcpyHostToDevice);
    return t;
  }
```

311 ⟨ Global functions 8 ⟩ +≡

```
  void transfer_tables_from_gpu(GPUTables *g, GPUTables *h)
  {
    void *t;
    cudaMemcpy(h, g, sizeof(GPUTables), cudaMemcpyDeviceToHost);
    t = h→ctab;
    h→ctab = mem_typed_alloc(h→nc, struct subcuboids_table_item,
        mem_phase_two);
    cudaMemcpy(h→ctab, t, h→nc * sizeof(struct subcuboids_table_item),
        cudaMemcpyDeviceToHost);
    t = h→s;
    h→s = mem_typed_alloc(h→ns, struct solids_table_item, mem_phase_two);
    cudaMemcpy(h→s, t, h→ns * sizeof(struct solids_table_item),
        cudaMemcpyDeviceToHost);
    t = h→p;
    h→p = mem_typed_alloc(h→np, struct primitives_table_item,
        mem_phase_two);
    cudaMemcpy(h→p, t, h→np * sizeof(struct primitives_table_item),
        cudaMemcpyDeviceToHost);
    t = h→pb;
    h→pb = mem_typed_alloc(h→npb, int32_t, mem_phase_two);
    cudaMemcpy(h→pb, t, h→npb * sizeof(int32_t), cudaMemcpyDeviceToHost);
    t = h→sb;
    h→sb = mem_typed_alloc(h→nsb, uint32_t, mem_phase_two);
    cudaMemcpy(h→sb, t, h→nsb * sizeof(uint32_t), cudaMemcpyDeviceToHost);
    t = h→ctree;
    h→ctree = mem_typed_alloc(h→nct, double, mem_phase_two);
    cudaMemcpy(h→ctree, t, h→nct * sizeof(double), cudaMemcpyDeviceToHost);
    t = h→iltab;
    h→iltab = mem_typed_alloc(MAX_SFIELD + 1, int8_t, mem_phase_two);
    cudaMemcpy(h→iltab, t, (MAX_SFIELD + 1) * sizeof(double),
        cudaMemcpyDeviceToHost);
    t = h→ntab;
    h→ntab = mem_typed_alloc(h→nc * NUM_NEIGHBOURS, uint32_t,
        mem_phase_two);
    cudaMemcpy2D(h→ntab, NUM_NEIGHBOURS * sizeof(uint32_t), t, h→pitch,
        NUM_NEIGHBOURS * sizeof(uint32_t), h→nc, cudaMemcpyDeviceToHost);
  }
```

312 ⟨ Global functions 8 ⟩ +≡

```
  void print_tables(FILE *f, GPUTables *g)
  {
    uint32_t i, j;
    fprintf(f, "G␣E␣O␣M␣E␣T␣R␣Y␣␣T␣A␣B␣L␣E\nPrimitives␣table:\n");
    for (i = 0; i < g→np; ++i) {
      fprintf(f, "Affine:\n");
      matrix_print(f, g→p[i].a, 4, 4, 0);
      fprintf(f, "\nInverse:\n");
      matrix_print(f, g→p[i].i, 4, 4, 0);
      for (j = 0; j < 80; ++j) fprintf(f, "-");
      fprintf(f, "\n");
    }
    fprintf(f, "Solids␣table:\n");
    for (i = 0; i < g→ns; ++i) fprintf(f, "%u␣%u\n", g→s[i].s, g→s[i].c);
    fprintf(f, "Postfix␣buffer:\n");
    for (i = 0; i < g→npb; ++i) fprintf(f, "%d␣", g→pb[i]);
    fprintf(f, "\nSubcuboids␣table:\n");
    for (i = 0; i < g→nc; ++i) fprintf(f, "%u␣%u\n", g→ctab[i].s, g→ctab[i].c);
    fprintf(f, "Solid␣indices␣buffer:\n");
    for (i = 0; i < g→nsb; ++i) fprintf(f, "%u␣", g→sb[i]);
    fprintf(f, "\n");
  }
```

313 ⟨ Test gpu tables 313 ⟩ ≡

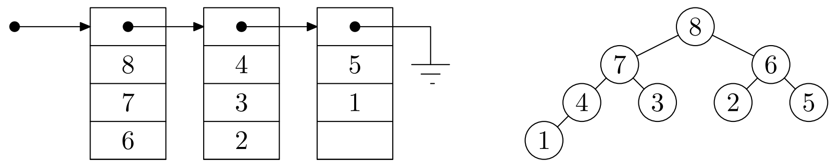
```
  {
    if (false ≡ read_geometry("test/test_gpu_table.data")) exit(1);
    print_forest( );
    create_geotab(&geotab);
    GPUTables *gpu, host;
    gpu = transfer_tables_to_gpu(&geotab);
    transfer_tables_from_gpu(gpu, &host);
    print_tables(stdout, &host);
  }
```

Part V

Particle repository

All of the particles yet to be simulated are stored in a particles repository. The particles are grouped according to their containing subcuboid, so that each group with the same solids can be simulated in one batch.

The particle repository is implemented as a paged binary *max-heap* (we could have use a *min-heap*, but this doesn't matter), which is allowed to grow. The heap is maintained as a linked list of heap pages, where each heap page is a fixed array of particles, with the first array element used as a pointer-to-a-pointer to maintain the linked list of heap pages.



The paging only becomes active when the first allocated heap page is insufficient to fulfill the required number of particles. Since paging adds computational overhead, it is expected that the first page is allocated carefully so as to avoid paging. Keeping this in mind, we provide two versions of all the heap functions. The functions with the 'fast' suffix does not assume paging, whereas, those with 'paged' suffix assume paging.

314
```
#define HEAP_PAGE_SIZE 4
            /* 4096 bytes per page, including linked list 'next' pointer */
⟨ Type definitions 6 ⟩ +≡
  typedef struct particle_struct Particle;
  typedef struct particle_repository_struct {
    uint32_t pid;    /* particle identifier to use next */
    uint32_t count;    /* current number of particles in heap */
    uint32_t max;    /* maximum number of particles allowed in heap */
    uint16_t page_size;    /* number of particles per page */
    Particle *head, *tail;    /* pointers to first and last pages */
  } ParticleRepository;
  ParticleRepository particles = {1, 0, 0, HEAP_PAGE_SIZE, Λ, Λ};
```

Function *heap_insert_fast*(*pr*, *t*) inserts a new particle *t* into the particles repository *pr*. This insertion does not assume heap paging. In the first page, we *bubble up* the new node to its rightful place using the containing subcuboid as the comparison key. This bubbling is done by climbing up the tree starting at the array position dictated by the array implementation of a complete binary tree, and moving the node from parent to child as we climb. The insertion has completed when the node is finally placed at its rightful place.

315
```
⟨ Global functions 8 ⟩ +≡
  void heap_insert_fast(ParticleRepository *pr, Particle *t)
```

```
{
  uint32_t p, n;    /* indices of parent and current node */
  Particle *fp;
  fp = pr→head;    /* first page */
  n = ++pr→count;    /* start bubbling from the last node */
  p = n ≫ 1;
  while (p > 0) {    /* bubble up the tree */
    if (fp[p].subcuboid < t→subcuboid) fp[n] = fp[p];
        /* move from parent to child */
    else break;    /* rightful place found */
    n = p;    /* climb up the tree */
    p = n ≫ 1;
  }
  if (0 ≡ t→id) t→id = pr→pid++;    /* give unique id to particle */
  fp[n] = *t;    /* place node */
}
```

Function *heap_remove_fast*(*pr*, *t*) removes the particle at the top of the max-heap representing the particles repository *pr*, and stores the particle into *t*. This removal does not assume heap paging. From the first page, we remove the first node, which is always the top of the heap. We then fill this void by promoting the last node of the complete binary tree to become the root. Finally, we rebalance the heap by *bubbling down* the root until we find its rightful place, or we find that it has becomes a leaf. While bubbling down, we move either the left or right subtree root to the current node as we climb down the tree.

316 ⟨ Global functions 8 ⟩ +≡

```
void heap_remove_fast(ParticleRepository *pr, Particle *t)
{
  uint32_t n, l, r, q;    /* indices to node, left, right and last */
  Particle temp, *fp;
  fp = pr→head;    /* first page */
  *t = fp[1];    /* particle to return */
  q = pr→count--;
  if (q) {
    temp = fp[q];    /* the last node to be promoted */
    n = 1;    /* start at the root */
    l = 2;
    while (l < q) {
        /* bubble down the tree choosing left or right subtree */
      r = l + 1;
      if (fp[q].subcuboid < fp[l].subcuboid) {
        if (r < q ∧ fp[l].subcuboid < fp[r].subcuboid) {
            fp[n] = fp[r];
            n = r;
          }
          else {
            fp[n] = fp[l];
            n = l;
          }
        }
        else {
          if (r < q ∧ fp[q].subcuboid < fp[r].subcuboid) {
            fp[n] = fp[r];
            n = r;
          }
          else break;     /* rightful place found */
        }
        l = n ≪ 1;     /* go deeper until we have passed a leaf */
      }
      fp[n] = temp;     /* place node */
    }
  }
```

For a given particle index n within the heap pr, function *heap_find_pidx*(pr, n, p, i) finds the page start address p and the index i within that page. The binary heap is implemented using an array representation of a complete binary tree. Hence, while maintaining this tree we are required to find the parent, or children, of a given tree node, and if paging is active, we are required to find its page and index within the heap.

317 ⟨ Global functions 8 ⟩ +≡

```
  void heap_find_pidx(ParticleRepository *pr, uint32_t n, Particle
          **p, uint32_t *i)
  {
    Particle *t = pr→head;
    while (n > pr→page_size) {
      t = (Particle *) *(char **) t;
      n −= pr→page_size;
    }
    *i = n;
    *p = t;
  }
```

Function *heap_insert_paged*(pr, t) inserts a new particle t into the particles repository pr. This insertion uses the same method as *heap_insert_fast*(pr, t), except that it assumes heap paging; hence, to reference, place, or move a node, we first find the correct page and index for each node using *heap_find_pidx*(*pr*, *n*, *p*, *i*).

318 ⟨ Global functions 8 ⟩ +≡

```
void heap_insert_paged(ParticleRepository *pr, Particle *t)
{
  uint32_t p, n;    /* indices of parent and current node within heap */
  Particle *p_pg, *n_pg;    /* heap pages of parent and current node */
  uint32_t p_idx, n_idx;
     /* indices within heap pages of parent and current node */
  n = ++pr→count;    /* start bubbling from the last node */
  p = n ≫ 1;
  while (p > 0) {    /* bubble up the tree */
    heap_find_pidx(pr, p, &p_pg, &p_idx);
    heap_find_pidx(pr, n, &n_pg, &n_idx);
    if (p_pg[p_idx].subcuboid < t→subcuboid) n_pg[n_idx] = p_pg[p_idx];
    else break;    /* rightful place found */
    n = p;    /* climb up the tree */
    p = n ≫ 1;
  }
  if (0 ≡ t→id) t→id = pr→pid++;    /* give unique id to particle */
  heap_find_pidx(pr, n, &n_pg, &n_idx);
  n_pg[n_idx] = *t;
}
```

Function *heap_remove_paged*(*pr*, *t*) removes a new particle *t* into the particles repository *pr*. This removal uses the same method as *heap_remove_fast*(*pr*, *t*), except that it assumes heap paging; hence, to reference, or change a node, we first find the correct page and index for each node using *heap_find_pidx*(*pr*, *n*, *p*, *i*).

319 ⟨ Global functions 8 ⟩ +≡

```
void heap_remove_paged(ParticleRepository *pr, Particle *t)
{
  uint32_t n, l, r, q;    /* indices to node, left, right and last */
  Particle *n_pg, *l_pg, *r_pg, *q_pg;    /* heap pages */
  uint32_t n_idx, l_idx, r_idx, q_idx;    /* indices within heap pages */
  Particle temp;
  *t = pr→head[1];    /* particle to return */
  q = pr→count−−;
  if (q) {
    heap_find_pidx(pr, q, &q_pg, &q_idx);
    temp = q_pg[q_idx];    /* the last node to be promoted */
    n = 1;    /* start at the root */
    $l = 2$;
    **while** $(l < q)$ {
          /* bubble down the tree choosing left or right subtree */
      $r = l + 1$;
      $heap\_find\_pidx(pr, l, \&l\_pg, \&l\_idx)$;
      $heap\_find\_pidx(pr, r, \&r\_pg, \&r\_idx)$;
      $heap\_find\_pidx(pr, n, \&n\_pg, \&n\_idx)$;
      **if** $(q\_pg[q\_idx].subcuboid < l\_pg[l\_idx].subcuboid)$ {
        **if** $(l\_pg[l\_idx].subcuboid < r\_pg[r\_idx].subcuboid)$ {
          $n\_pg[n\_idx] = r\_pg[r\_idx]$;
          $n = r$;
        }
        **else** {
          $n\_pg[n\_idx] = l\_pg[l\_idx]$;
          $n = l$;
        }
      }
      **else** {
        **if** $(q\_pg[q\_idx].subcuboid < r\_pg[r\_idx].subcuboid)$ {
          $n\_pg[n\_idx] = r\_pg[r\_idx]$;
          $n = r$;
        }
        **else break**;     /* rightful place found */
      }
      $l = n \ll 1$;     /* go deeper until we have passed a leaf */
    }
    $heap\_find\_pidx(pr, n, \&n\_pg, \&n\_idx)$;
    $n\_pg[n\_idx] = temp$;     /* place node */
  }
}
```

The following are the return codes that must be checked by the caller of the following functions.

320 ⟨ Type definitions 6 ⟩ +≡

```
  **enum** {
    HEAP_SUCCESS = 0, HEAP_EMPTY, HEAP_FULL, HEAP_ERROR_ALLOC,
        HEAP_ERROR_UNDEFINED
  };
```

Function *heap_expand*(r) expands the heap r by adding a new page at the end of the linked list.

321 ⟨ Global functions 8 ⟩ +≡

```
int heap_expand(ParticleRepository *r)
{
  Particle *t = mem_typed_alloc(HEAP_PAGE_SIZE + 1, Particle,
        mem_phase_two);
  if (Λ ≡ t) return HEAP_ERROR_ALLOC;
  *(char **) t = Λ;      /* make last page: 'next' points to Λ */
  r→max += r→page_size;
  *((char **) r→tail) = (char *) t;
  r→tail = t;
  return HEAP_SUCCESS;
}
```

Function *heap_insert*(r, p, e) inserts a new particle p into the particle repository r. If e is *true* and there is not enough space in the heap, the heap will be expanded to fit p.

The macro *heap_has_space*(r) may be used to check if there is empty space in r before making an insertion call.

322 **#define** *heap_has_space*(r) $((r).count < (r).max)$

⟨ Global functions 8 ⟩ +≡

```
int heap_insert(ParticleRepository *r, Particle *p, bool e)
{
  int i;
  if (r→count < r→max) {
    if (r→count > r→page_size) heap_insert_paged(r, p);
    else heap_insert_fast(r, p);
  }
  else {
    if (e) {
      i = heap_expand(r);
      if (i) return i;
      heap_insert_paged(r, p);
    }
    else return HEAP_FULL;
  }
  return HEAP_SUCCESS;
}
```

Function *heap_remove*(r, t) removes the particle at the top of the max-heap representing the particles repository r, and stores the particle into p.

323 ⟨ Global functions 8 ⟩ +≡
```
int heap_remove(ParticleRepository *r, Particle *p)
{
  if (r→count ≡ 0) return HEAP_EMPTY;
  if (r→count > r→page_size) heap_remove_paged(r, p);
  else heap_remove_fast(r, p);
  return HEAP_SUCCESS;
}
```

Function *heap_init*(r) initialises the max-heap r by creating the first page.

324 ⟨ Global functions 8 ⟩ +≡
```
int heap_init(ParticleRepository *r)
{
  r→head = mem_typed_alloc(HEAP_PAGE_SIZE + 1, Particle, mem_phase_two);
  if (Λ ≡ r→head) return HEAP_ERROR_ALLOC;
  r→count = 0;
  r→pid = 1;
  r→max = r→page_size = HEAP_PAGE_SIZE;
  *(char **) r→head = Λ;
  r→tail = r→head;
  return HEAP_SUCCESS;
}
```

Function *heap_print*(f, r) prints the heap r to the I/O stream pointed to by f.

325 ⟨ Global functions 8 ⟩ +≡
```
void heap_print(FILE *f, ParticleRepository *r)
{
  int i, j, p;
  Particle *page;
  fprintf(f, "Number␣of␣particles:␣%u", r→count);
  p = 0;
  j = r→count;
  page = r→head;
  while (page) {
    if (j) {
      fprintf(f, "\nPage␣%u:␣", p);
      for (i = 1; j ∧ i ≤ r→page_size; ++i, --j)
        fprintf(f, "%u␣", page[i].subcuboid);
      page = (Particle *) *(char **) page;
      p++;
    }
```

```
        else break;
      }
      fprintf(f, "\n");
    }
```

326 ⟨ Test particle repository 326 ⟩ ≡

```
    {
      Particle p;
      heap_init(&particles);
      do {
        printf("Enter␣subcuboid␣(enter␣0␣to␣remove␣from␣heap):␣");
        scanf("%u", &(p.subcuboid));
        if (p.subcuboid ≡ 0) {
          if (heap_remove(&particles, &p) ≠ HEAP_SUCCESS) break;
          printf("%u\n", p.subcuboid);
        }
        else if (heap_insert(&particles, &p, true) ≠ HEAP_SUCCESS) break;
        heap_print(stdout, &particles);
      } while (1);
    }
```

Part VI

Simulation

The particle simulations are carried out in batches. After a simulation has begun, it will only exit under two conditions:

1) All of the particles have been processed. This includes all of the primary particles generated by the particle gun, and all of the secondary particles generated by physics processes.

2) There was an error while processing the particles. There are several causes for error, for instance, there is not enough space to accommodate the heap expansion required for adding secondary particles, etc.

3.11 Particle

A *particle* is the lowest-level data structure that `MCS` manipulates. This data structure represents a fundamental particle in physics, which interacts with the materials in the world.

In `MCS`, all of the primary particles to be simulated are generated by the particle gun. The origin of these particles are specified by the vertex used by the particle gun. A particle gun can generate particles from various vertices.

328 ⟨Type definitions 6⟩ +≡
```
struct particle_struct {
    ⟨Physical properties of a particle 329⟩;
    ⟨Auxilliary data for managing a particle 330⟩;
};
```

These are the properties used by the physics processes. Some of these must be initialised using a randomiser, thus satisfying the principle requirement for a Monte Carlo simulation.

329 ⟨Physical properties of a particle 329⟩ ≡
```
Vector v;   /* position */
Vector mo;   /* momentum */
Vector po;   /* polarisation */
double m;   /* mass */
double c;   /* charge */
```
This code is used in chunk 328.

Every particle knows which subcuboid it is currently inside. It also maintains a unique identifier, which is supplied by the particle repository. Finally, to determine the particle hierarchy, each particle also maintains the identifier of its parent and a sibling index, which ranks the children of a particle.

330 ⟨Auxilliary data for managing a particle 330⟩ ≡
```
uint32_t subcuboid;   /* index of subcuboid containing particle */
uint32_t id;   /* particle identifier */
uint32_t pi;   /* parent identifier */
uint16_t si;   /* sibling index (rank among siblings) */
uint16_t nd;   /* number of daughter particles */
uint8_t s;   /* the s-field */
uint8_t active;   /* particle type */
```
This code is used in chunk 328.

```
331 ⟨ Print particle information to stdout 331 ⟩ ≡
  fprintf(stdout,
      "%u␣(%lf,␣%lf,␣%lf)␣%u␣(%lf,␣%lf,␣%lf)␣""(%lf,␣%lf,␣%lf)␣%lf\
      ␣%lf␣%u␣%u␣%u\n", p→id, p→v[0], p→v[1], p→v[2], p→s, p→mo[0], p→mo[1],
      p→mo[2], p→po[0], p→po[1], p→po[2], p→m, p→c, p→pi, p→si, p→nd);
```

```
332 ⟨ Print particle information to gpfile 332 ⟩ ≡
  fprintf(gpfile, "%6u␣(%lf,␣%lf,␣%lf)␣%u␣(%lf,␣%lf,␣%lf)␣\
      ""(%lf,␣%lf,␣%lf)␣%lf␣%lf␣%u␣%u␣%u\n", gpbuff[i].id, gpbuff[i].v[0],
      gpbuff[i].v[1], gpbuff[i].v[2], gpbuff[i].s, gpbuff[i].mo[0], gpbuff[i].mo[1],
      gpbuff[i].mo[2], gpbuff[i].po[0], gpbuff[i].po[1], gpbuff[i].po[2], gpbuff[i].m,
      gpbuff[i].c, gpbuff[i].pi, gpbuff[i].si, gpbuff[i].nd);
```

This code is used in chunk 334.

Function *save_particle*(p) saves particle p in the buffer to be written to the filesystem, evetually. When a particle is no longer active, they must be transferred to the filesystem for later analysis. To reduce the number of system calls, we use a particles buffer to hold particles until they can be written in a batch write to the file system. We maintain the buffer, the I/O stream and the number of particles as global variables. These must be initialised appropriately before calling *run_simulation*( ).

```
333 #define MAX_PARTICLE_BUFFER 60
            /* number of particles in output buffer */
⟨ Global functions 8 ⟩ +≡
  Particle gpbuff[MAX_PARTICLE_BUFFER];
  uint32_t gpbuffsize = 0;    /* number of particles in buffer */
  FILE *gpfile = Λ;    /* the I/O stream to write particle data */
  void save_particle(Particle *p)
  {
    if (gpbuffsize ≡ MAX_PARTICLE_BUFFER)
      ⟨ Flush particles in buffer to filesystem 334 ⟩;
    gpbuff[gpbuffsize++] = *p;
  }
```

```
334 ⟨ Flush particles in buffer to filesystem 334 ⟩ ≡
  {
    uint32_t i;
    for (i = 0; i < gpbuffsize; ++i) ⟨ Print particle information to gpfile 332 ⟩;
    gpbuffsize = 0;
  }
```

This code is used in chunks 333 and 359.

3.12 Vertex

To generate particles, a particle gun must first be placed inside the world by choosing a vertex. When the particle gun is activated, primary particles are generated from this vertex. A vertex is not associated with a particle gun, however; they are stored with respect to an event. Hence, an event can have multiple vertices associated with it. These are stored as a linked-list.

335 ⟨ Type definitions 6 ⟩ +≡
```
typedef struct vertex_struct {
  Vector v;     /* particle gun position vector */
  Vector mo;     /* momentum */
  Vector po;     /* polarisation */
  double e;     /* energy */
  double c;     /* charge */
  uint32_t np;     /* number of particles required */
  struct vertex_struct *next;
} Vertex;
```

Function *create_vertex*(*e*) creates a new vertex under the supplied event *e*. This functions returns a pointer to the vertex, or Λ if a new vertex could not be created.

336 **#define** *vertices_per_block* 10 /* TODO: */

⟨ Global functions 8 ⟩ +≡
```
static Vertex *next_vertex = Λ;
static Vertex *bad_vertex = Λ;
Vertex *create_vertex(Event *e)
{
  Vertex *v = next_vertex;
  if (v ≡ bad_vertex) {
    v = mem_typed_alloc(vertices_per_block, Vertex, mem_phase_two);
    if (Λ ≡ v) return Λ;
    else {
      next_vertex = v + 1;
      bad_vertex = v + vertices_per_block;
    }
  }
  else ++next_vertex;
  ⟨ Insert vertex to event 337 ⟩;
  return v;
}
```

337 ⟨ Insert vertex to event 337 ⟩ ≡

```
  e->nv++;
  v->next = e->v;
  e->v = v;
```
This code is used in chunk 336.

3.13 Events

An *event* is the highest-level simulation object. It provides a link between the user and the simulator. Users specify the number of events they wish to simulate. An event is associated with various vertices which specify the locations where a particle gun could be placed. To generate primary particles, the particle gun must first be placed on one of the vertices of the event being simulated.

```
338 ⟨ Type definitions 6 ⟩ +≡
  typedef struct event_struct {
    uint32_t id;    /* event identifier */
    uint32_t nv;    /* number of vertices */
    Vertex *v;    /* head of the vertex linked list */
    struct event_struct *next;    /* pointer to next event */
  } Event;
```

Function *create_event*( ) creates a new event with a unique identifier. This functions returns a pointer to the event, or Λ if a new event could not be created.

```
339 #define events_per_block 10    /* TODO: */
⟨ Global functions 8 ⟩ +≡
  static Event *next_event = Λ;
  static Event *bad_event = Λ;
  static uint32_t next_event_id = 0;
  Event *create_event( )
  {
    Event *e = next_event;
    if (e ≡ bad_event) {
      e = mem_typed_alloc(events_per_block, Event, mem_phase_two);
      if (Λ ≡ e) return Λ;
      else {
        next_event = e + 1;
        bad_event = e + events_per_block;
      }
    }
    else ++next_event;
    e→id = ++next_event_id;
    return e;
  }
```

3.14 Particle gun

340 ⟨ Generate primary particle using particle gun 340 ⟩ ≡

```
  vector_zero(p.v);
  vector_zero(p.mo);
  vector_zero(p.po);
  p.m = p.c = 0.0;
  p.id = p.pi = p.si = 0;
  p.active = true;     /* only active particles inside heap */
  p.s = #0;     /* renew: s-field are changed by physics processes */
  p.nd = 0;     /* no daughters yet */
  p.subcuboid = find_subcuboid(geotab.ctree, p.v);
```

This code is used in chunk 348.

3.15 Particle simulation

Function *simulate_particle*(p, j, s, k) simulates the particle in $p[j]$ by applying the physics processes. The resulting particle is stored in $s[k]$. All of the secondary particles generated by the physics processes are then stored in the subsequent array elements following $s[k]$. This function returns the next valid array element in s, which will be used in the next call to *simulate_particle*(p, j, s, k).

```
342 ⟨Global functions 8⟩ +≡
  uint32_t simulate_particle(Particle *p, uint32_t j, Particle *s, uint32_t k)
  {
    uint32_t solid;
    bool found_solid = false;
    ⟨Find the solid which contains the particle 343⟩;
    ⟨Find the material properties of the containing solid 344⟩;
    ⟨Apply physics processes 345⟩;
    ⟨Update secondary particles array 346⟩;
    return k;
  }
```

```
343 ⟨Find the solid which contains the particle 343⟩ ≡
  {
    uint32_t sc, start, end;
    sc = p[j].subcuboid;
    start = geotab.ctab[sc].s;
       /* lookup subcuboids table and retrieve start index */
    end = start + geotab.ctab[sc].c;
       /* and count to the solid indices buffer */
    while (start < end) {
      solid = geotab.sb[start];
         /* lookup solid indices buffer and retrieve solid index */
      if (solid_contains_particle(solid, &p[j])) {
        found_solid = true;
        break;
      }
      ++start;
    }
    info("\t\tFound␣solid:␣%d\n", solid);
  }
```
This code is used in chunk 342.

```
344 ⟨Find the material properties of the containing solid 344⟩ ≡
```
This code is used in chunk 342.

345 ⟨Apply physics processes 345⟩ ≡
This code is used in chunk 342.

346 ⟨Update secondary particles array 346⟩ ≡
This code is used in chunk 342.

3.16 Batch simulation

All of the particle simulations are run in batches of particle blocks. The following global variables are used to maintain the current batch and block information.

```
347 ⟨ Global variables 4 ⟩ +≡
  uint16_t current_batch = 0;    /* current batch */
```

Function *generate_primaries*( ) fills up the particles repository with primary particles generated using a particle gun. It returns zero if all of the primary particles required by the event vertices have already been processed; otherwise, the number of new primaries are returned. The batch simulation loop will continue to be valid as long as the return value of this function is nonzero.

While generating primaries, the aim is to fill up the max-heap without expanding the heap, so that only primary particles that are needed to fill up a batch are generated. Of course, since this function will be called multiple times by *get_particle*( ) while creating a simulation batch, all of the particles that are required by all of the vertices in all of the events will be processed, eventually.

During multiple calls to *generate_primaries*( ), we maintain from one call to the next the current event, the current vertex within that event, and the number of particles already generated for the current vertex. Hence, before running a simulation, i.e., calling *run_simulation*( ), these global variables must be initialised appropriately.

```
348 ⟨ Global functions 8 ⟩ +≡
  Event *ge = Λ;    /* current simulation event */
  Vertex *gv = Λ;    /* current vertex within simulation event */
  uint32_t gc = 0;    /* current particle count for current vertex */
  int generate_primaries( )
  {
    Particle p;    /* primary particle to be generated */
    int n = 0;    /* number of particles generated in this call */
    while (ge) {
      while (gv) {
        while (gc < gv→np) {
          if (heap_has_space(particles)) {
            ⟨ Generate primary particle using particle gun 340 ⟩;
            heap_insert(&particles, &p, false);
            ++n;
            ++gc;
          }
          else goto heap_is_full;
        }
        gv = gv→next;    /* move to the next vertex */
        gc = 0;
      }
      ge = ge→next;      /* move to the next event */
      if (ge) {
        gv = ge→v;
        gc = 0;
      }
    }
  heap_is_full: info("\tGenerated␣%d␣primary␣particles\n", n);
    return n;
  }
```

Function *get_particle*(*p*) retrieves a particle from the particle repository and stores the particle in *p*. It returns 1 if a particle was retrieved successfully; otherwise, returns 0. If there are no particles in the particles repository, it will first attempt to fill up the particles repository, by generating primary particles using the particle gun, and then retry to retrieve particle from the heap.

349 ⟨ Global functions 8 ⟩ +≡

```
  int get_particle(Particle *p)
  {
    if (HEAP_EMPTY ≡ heap_remove(&particles, p)) {
      if (generate_primaries()) heap_remove(&particles, p);      /* try again */
      else return 0;      /* no more particles left */
    }
    return 1;      /* one particle retrieved */
  }
```

All of the primary particles that was added from the particles repository are stored in the particles array *p*, whereas, the particles generated as secondaries by the compute threads as a result of physics processes are stored in the particles array *s*. Before the compute threads have worked on the block, the number of secondaries *ns* are always set to zero by the function *create_batch*( ). After the block has been processed, the primary particles array *p* could be empty; however, the secondary particles array *s* will never be empty.

350
```
#define MAX_PRIMARIES_BLOCK 4      /* must depend on available memory */
#define MAX_DAUGHTERS 4      /* maximum number of daughters allowed */
#define MAX_SECONDARIES_BLOCK (MAX_DAUGHTERS * MAX_PRIMARIES_BLOCK)
```

⟨ Type definitions 6 ⟩ +≡

```
  typedef struct block_struct {
    uint32_t subcuboid;      /* index of the associated subcuboid */
    uint16_t np;      /* number of primaries */
    uint16_t ns;      /* number of secondaries */
    Particle p[MAX_PRIMARIES_BLOCK];
    Particle s[MAX_SECONDARIES_BLOCK];
  } Block;
```

Each simulation batch comprises of several simulation blocks that contain data about the particles and the subcuboid which contains those particles. Each of these blocks could be processed exclusively by a single thread, or a group of compute threads.

We have used fixed-size arrays, instead of dynamically allocated memory blocks, so that the input data as they are available on the host application, or the output data as they are available in the compute memory, could be trasferred using a single memory copy of the batch. For instance, if we wish to use multiple threads that are available on graphics processing units, we can easily transfer the batches back and forth using a single memory copy between the host application and the graphics hardware. This is also important because, for cache efficiency, we required the memory footprint on the compute devices to be quite simple and compact.

351
```
#define MAX_BLOCKS_BATCH  4
            /* must depend on available compute blocks */
⟨ Type definitions 6 ⟩ +≡
  typedef struct sim_batch_struct {
    uint16_t nb;      /* number of blocks with particles in them */
    uint32_t np;      /* number of particles in all of the blocks */
    Block b[MAX_BLOCKS_BATCH];      /* array of blocks */
  } Batch;
```

Function *create_batch*(*b*) creates a simulation batch and stores the batch in *b*. If successful, it returns a positive number that gives the number of particles in the batch; otherwise, it returns zero to mark a successful end of simulation.

A simulation batch is created by starting at the first slot of the first block. We continue filling each slot in each of the blocks by taking particles from the particles repository, until all of the blocks are full. Furthermore, since all of the particles in a block must be contained by the same subcuboid that is associated with that block, we move forward to the next block when the particle retrieved from the particles repository has a smaller subcuboid index than the subcuboid index associated with the current block. Since the particles repository is implemented as a max-heap, where the subcuboid index is used as a comparison key, the *get_particle*( ) call will never return a particle with subcuboid index greater than the subcuboid index of the current block. The blocks, therefore, effectively group the particles that belong in the same subcuboid.

```
352 ⟨ Global functions 8 ⟩ +≡
  int create_batch(Batch *b)
  {
    Particle p;     /* particle retrieved from heap */
    uint32_t i, j;     /* current block, and current slot within block */
    i = j = 0;     /* start at first slot of first block */
    b->nb = b->np = 0;     /* reset block and particle count */
    info("Creating␣simulation␣batch␣%d\n", current_batch);
    if (get_particle(&p)) {
      b->b[i].subcuboid = p.subcuboid;     /* subcuboid index for the block */
      b->b[i].p[j++] = p;     /* add particle to block */
      ++b->nb;     /* increment number of active blocks */
      ++b->np;     /* increment number of particles in batch */
      while (get_particle(&p)) {
        if (b->b[i].subcuboid ≠ p.subcuboid ∨ MAX_PRIMARIES_BLOCK ≡ j) {
              /* change block */
          b->b[i].np = j;     /* finalise current block */
          b->b[i].ns = 0;
          if (MAX_BLOCKS_BATCH ≡ ++i) {     /* move to next block */
            heap_insert(&particles, &p, false);
                /* batch is full: retain particle for next batch */
            goto batch_is_full;
          }
          b->b[i].subcuboid = p.subcuboid;
              /* subcuboid index for the new block */
          ++b->nb;     /* increment number of active blocks */
          j = 0;     /* first slot of new block */
        }
        b->b[i].p[j++] = p;     /* add particle to block */
        ++b->np;     /* increment number of particles in batch */
      }
      b->b[i].np = j;     /* finalise current block */
      b->b[i].ns = 0;
    }
  batch_is_full: return b->np;
  }
```

Function *update_repository*(*b*) updates the max-heap, following a successful simulation of the batch *b*, by moving primary and secondary particles from the active blocks in that batch to the particles repository.

```
353 ⟨ Global functions 8 ⟩ +≡
  int update_repository(Batch *b)
```

```
{
  uint16_t i, j;
  Particle p;
  for (i = 0; i < b->nb; ++i) {
    ⟨ Move unprocessed primary particles to repository 354 ⟩;
    ⟨ Process particles in the secondary particles array after a batch
        simulation 355 ⟩;
  }
  return 0;
}
```

It is unnecessary and inefficient to move primary particles from the block back to the particles repository. However, for now, we take the simpler route. Future changes must attempt to update the blocks without having to removed unprocessed primary particles that are already in the block.

354 ⟨ Move unprocessed primary particles to repository 354 ⟩ ≡

```
  for (j = 0; j < b->b[i].np; ++j) heap_insert(&particles, &b->b[i].p[j], true);
```

This code is used in chunk 353.

After a block has been processed by the compute threads, all of the particles in the block's secondary particles array s are of three types: 1) primary particles that are no longer active, 2) primary particles that are still active, and 3) daughter particles generated by the physics processes. All of these particles must first be saved to the file system for later analysis. In the second and third cases, we prepare the particle for further processing, by updating its containing subcuboid using the s-field. After this, we only move the particle to the particles repository if it continues to stay inside the simulation world.

355 ⟨ Process particles in the secondary particles array after a batch simulation 355 ⟩ ≡

```
  for (j = 0; j < b->b[i].ns; ++j) {
    p = b->b[i].s[j];
    save_particle(&p);
    if (p.active) {
      ⟨ Prepare particle for further processing 356 ⟩;
      if (OUTSIDE_WORLD ≡ p.subcuboid) continue;
      heap_insert(&particles, &p, true);
    }
  }
```

This code is used in chunk 353.

After a primary particle has been processed, or a secondary particle has been generated by physics processes, the particle's position vector v reflects its new location. Using this new location, and the bounding box bb of the containing subcuboid, we must first update the particles s-field. The s-field is then used in combination with the index of the current subcuboid to determine the index of the new subcuboid that will now contain the particle. Once this is done, we reset the s-field for the next simulation batch.

```
356 ⟨ Prepare particle for further processing 356 ⟩ ≡
  update_sfield(&(p.s), &(geotab.ctab[p.subcuboid].bb), p.v);
  p.subcuboid = get_neighbour(&geotab, p.subcuboid, p.s);
  p.s = #0;      /* renew particle */
  p.id = 0;      /* get a new identifier from particle repository */
```

This code is used in chunk 355.

```
357 ⟨ Global functions 8 ⟩ +≡
  int simulate_batch(Batch *b)
  {
    uint32_t i = 0, j, k;
    info("Simulating␣batch␣%d␣with␣%d␣particles\n", current_batch, b→np);
    while (i < b→nb) {
      info("\tSimulating␣block␣%d␣with␣%d␣particles\n", i, b→b[i].np);
      j = 0;
      k = 0;
      while (j < b→b[i].np) {
           /* b-¿b[i].s[k] = b-¿b[i].p[j]; b-¿b[i].s[k].active = false; */
        k = simulate_particle(b→b[i].p, j, b→b[i].s, k);
        ++j;
      }
      b→b[i].ns = k;
      b→b[i].np = 0;
      info("\tBlock␣%d␣simulated\n", i);
      ++i;
    }
    info("Batch␣%d␣simulated\n\n", current_batch++);
    return 0;
  }
```

Function *run_simulation*( ) runs the simulation iteratively by creating and simulating batches in each iteration, until all of the particles, both primary and secondary, have been processed. It returns 0 if the simulation was a success; otherwise, a non-zero value is returned.

358 ⟨ Global functions 8 ⟩ +≡
```
int run_simulation( )
{
  int i = 0;
  Batch b;
  info("Running␣simulations␣now...\n");
  current_batch = 0;
  while ((i = create_batch(&b))) {
    if (i < 0) goto handle_error;
    if (simulate_batch(&b)) goto handle_error;
    else if (update_repository(&b)) goto handle_error;
  }
  info("Completed␣simulating␣%d␣batches␣without␣errors\n\n",
      current_batch);
  return 0;     /* simulation done */
handle_error:
  info("Completed␣simulating␣%d␣batches␣with␣errors!\n\n",
      current_batch);
  return 1;
}
```

359 ⟨ Test simulation batch 359 ⟩ ≡
```
{
  Event *e, *ee;
  Vertex *v, *vv;
  verbose = true;     /* initialise the geometry table */
  if (false ≡ read_geometry("test/test_gpu_table.data")) exit(1);
  print_geom_statistics(stdout);
  create_geotab(&geotab);
  print_geotab(stdout, &geotab);     /* first event */
  ee = create_event( );
  vv = create_vertex(ee);
  vv→np = 5;
  v = create_vertex(ee);
  v→np = 5;
  v = create_vertex(ee);
  v→np = 5;
  v = create_vertex(ee);
  v→np = 5;     /* second event */
  e = create_event( );
  v = create_vertex(e);
  v→np = 5;
```

```
    v = create_vertex(e);
    v→np = 5;
    v = create_vertex(e);
    v→np = 5;
    v = create_vertex(e);
    v→np = 5;
    e→next = ee;
    ee = e;    /* third event */
    e = create_event( );
    v = create_vertex(e);
    v→np = 5;
    v = create_vertex(e);
    v→np = 5;
    v = create_vertex(e);
    v→np = 5;
    v = create_vertex(e);
    v→np = 5;
    e→next = ee;
    ee = e;
    gpfile = stdout;
    heap_init(&particles);
    ge = ee;
    if (ge) {
      gv = ge→v;
      gc = 0;
    }
    run_simulation( );
    ⟨Flush particles in buffer to filesystem 334⟩;
  }
```

This code is used in chunk 360.

Part VII

The main program

360 ⟨ Preprocessor definitions ⟩
```
⟨ Include preamble for applications 361 ⟩
    int main(int argc, char *argv[])
    {
      ⟨ Test simulation batch 359 ⟩;
      ⟨ Clean up the system 366 ⟩;
      return 0;
    }
```

361 ⟨ Include preamble for applications 361 ⟩ ≡
```
⟨ Include system libraries 367 ⟩
⟨ Type definitions 6 ⟩
⟨ Forward declare functions 3 ⟩
⟨ Global variables 4 ⟩
⟨ Global functions 8 ⟩
```
This code is used in chunk 360.

362 ⟨ Parse command line arguments 362 ⟩ ≡

363 ⟨ Process input files 363 ⟩ ≡

364 ⟨ Create physics tables 364 ⟩ ≡

365 ⟨ Create and process events 365 ⟩ ≡

366 ⟨ Clean up the system 366 ⟩ ≡
```
mem_free(mem_phase_one);
cuda_mem_free(mem_gpu);      /* call before freeing mem_phase_two */
mem_free(mem_phase_two);
```
This code is used in chunk 360.

367 ⟨ Include system libraries 367 ⟩ ≡
```
#include <math.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "cuda_runtime_api.h"
```
This code is used in chunk 361.

368 ⟨ Global variables 4 ⟩ +≡
Vector *positive_xaxis_unit_vector* = {1.0, 0.0, 0.0, 1.0};

Part VIII

Index

Index

List of Refinements

⟨ Add new block to the area by updating the linked list 38 ⟩ Used in chunk 35.
⟨ Adjust index of the neighbour cuboid along the x-axis 253 ⟩ Used in chunk 252.
⟨ Adjust index of the neighbour cuboid along the y-axis 254 ⟩ Used in chunk 252.
⟨ Adjust index of the neighbour cuboid along the z-axis 255 ⟩ Used in chunk 252.
⟨ Alert error while reading file 85 ⟩ Used in chunk 84.
⟨ Alert failure to create operator node 87 ⟩ Used in chunk 84.
⟨ Alert failure to create parameter node 88 ⟩ Used in chunk 84.
⟨ Alert failure to create primitive solid 86 ⟩ Used in chunk 84.
⟨ Alert invalid division of simulation world 91 ⟩ Used in chunk 84.
⟨ Alert solid already exists 89 ⟩ Used in chunk 84.
⟨ Alert solid does not exists 90 ⟩ Used in chunk 84.
⟨ Allocate memory for the three subcuboid search trees 240 ⟩ Used in chunk 239.
⟨ Allocate space as a consecutive array of 256 bytes each 37 ⟩ Used in chunk 35.
⟨ Apply affine transformations to the corners 171 ⟩ Used in chunk 169.
⟨ Apply physics processes 345 ⟩ Used in chunk 342.
⟨ Auxilliary data for managing a particle 330 ⟩ Used in chunk 328.
⟨ Build tables and search trees for managing the subcuboids 257 ⟩ Used in chunk 272.
⟨ Calculate γ subtended by v on the xz-plane 211 ⟩ Used in chunk 209.
⟨ Calculate affine transformed bounding box 169 ⟩ Used in chunk 168.
⟨ Calculate axis aligned bounding box of the transformed bounding box 172 ⟩ Used in chunk 169.
⟨ Calculate bounding box of difference 176 ⟩ Used in chunk 173.
⟨ Calculate bounding box of intersection 175 ⟩ Used in chunk 173.
⟨ Calculate bounding box of primitive block 162 ⟩ Used in chunk 161.
⟨ Calculate bounding box of primitive cylinder 164 ⟩ Used in chunk 161.
⟨ Calculate bounding box of primitive sphere 163 ⟩ Used in chunk 161.
⟨ Calculate bounding box of primitive torus 165 ⟩ Used in chunk 161.

⟨ Calculate bounding box of union 174 ⟩ Used in chunk 173.
⟨ Calculate containment range for the block 198 ⟩ Used in chunk 99.
⟨ Calculate containment range for the cylinder 202 ⟩ Used in chunk 107.
⟨ Calculate distance of the two-dimensional xz-projection 204 ⟩ Used in chunk 203.
⟨ Calculate end angles for the torus 207 ⟩ Used in chunk 111.
⟨ Calculate radial containment range for the torus 206 ⟩ Used in chunk 111.
⟨ Calculate radial distance of v from *tube_center* 214 ⟩ Used in chunk 212.
⟨ Calculate the eight corners of the bounding box 170 ⟩ Used in chunk 169.
⟨ Calculate the neighbour using the axes bit-pairs 252 ⟩ Used in chunk 251.
⟨ Calculate the projected distance δ of v on the xz-plane 210 ⟩ Used in chunk 209.
⟨ Calculate the radial angle of v on the cross-section at *tube_center* 215 ⟩ Used in chunk 212.
⟨ Calculate vector *tube_center* on the center of the tube at γ 213 ⟩ Used in chunk 212.
⟨ Check if the CSG expression was valid and retrieve the result 298 ⟩ Used in chunk 293.
⟨ Check if the requested size of the block is valid 36 ⟩ Used in chunk 35.
⟨ Check if there are stray node 276 ⟩ Used in chunk 272.
⟨ Check if v is on the surface of the tube 216 ⟩ Used in chunk 209.
⟨ Check if v is outside the tube 212 ⟩ Used in chunk 209.
⟨ Check that the target does not already exists 114 ⟩ Used in chunks 113, 125, and 129.
⟨ Clean up the system 366 ⟩ Used in chunk 360.
⟨ Cleanup resources allocated to invalid geometry 92 ⟩ Used in chunk 84.
⟨ Counters used during geometry table generation 271 ⟩ Used in chunk 269.
⟨ Create a new primitive solid 95 ⟩ Used in chunks 94, 103, 105, and 109.
⟨ Create and process events 365 ⟩
⟨ Create and register rotation operator node 146 ⟩ Used in chunk 142.
⟨ Create and register scaling operator node 152 ⟩ Used in chunk 148.
⟨ Create and register translation operator node 138 ⟩ Used in chunk 133.
⟨ Create difference operator node 130 ⟩ Used in chunk 129.
⟨ Create intersection operator node 126 ⟩ Used in chunk 125.
⟨ Create new difference operator node 131 ⟩ Used in chunk 130.
⟨ Create new intersection operator node 127 ⟩ Used in chunk 126.
⟨ Create new union operator node 120 ⟩ Used in chunk 116.
⟨ Create physics tables 364 ⟩
⟨ Create rotation parameter node 145 ⟩ Used in chunk 142.
⟨ Create scaling parameter node 151 ⟩ Used in chunk 148.
⟨ Create translation parameter node 136 ⟩ Used in chunk 133.
⟨ Create union operator node 116 ⟩ Used in chunk 113.
⟨ Data for primitive block 47 ⟩ Used in chunk 44.
⟨ Data for primitive cylinder 51 ⟩ Used in chunk 44.
⟨ Data for primitive sphere 49 ⟩ Used in chunk 44.

⟨ Set target, parameters and parent pointers 140 ⟩ Used in chunks 138, 146, and 152.
⟨ Set up the matrix for rotation 224 ⟩ Used in chunk 145.
⟨ Set up the matrix for scaling 225 ⟩ Used in chunk 151.
⟨ Set up the matrix for translation 222 ⟩ Used in chunk 136.
⟨ Sizes of the geometry table components 270 ⟩ Used in chunk 269.
⟨ Subcuboids related data structures inside the geometry table 232, 235, 249 ⟩ Used in chunk 269.
⟨ Test containment after affine transformations 221 ⟩ Cited in chunk 217. Used in chunk 219.
⟨ Test containment in subtrees using boolean operators 220 ⟩ Used in chunk 219.
⟨ Test containment inside primitive solid 217 ⟩ Used in chunk 219.
⟨ Test geometry input 226 ⟩
⟨ Test geometry table generation 286 ⟩
⟨ Test gpu tables 313 ⟩
⟨ Test particle repository 326 ⟩
⟨ Test simulation batch 359 ⟩ Used in chunk 360.
⟨ Test subcuboid functionalities 260 ⟩
⟨ Transfer the last page of the solid indices buffer 282 ⟩ Used in chunk 281.
⟨ Transfer the remaining pages of the solid indices buffer 283 ⟩ Used in chunk 281.
⟨ Translation parameters 132 ⟩ Used in chunk 66.
⟨ Type definitions 6, 11, 24, 32, 43, 44, 57, 58, 59, 69, 196, 231, 266, 267, 269, 287, 301, 309, 314, 320, 328, 335, 338, 350, 351 ⟩ Used in chunk 361.
⟨ Update secondary particles array 346 ⟩ Used in chunk 342.
⟨ Validate the test-case 229 ⟩ Used in chunk 227.