Surface Area Heuristic

The Surface Area Heuristic (SAH) of a split is the potential cost of traversal and intersection for a given split. The SAH itself provides no way of finding the minimal costs. Luckily the minima can only be found on minimum and maximum bounds of an object within the voxel being split. This is because the change in cost is linear between each edge bound of an object. It is then easy to sweep across all possible minima with only O(n log n) complexity. While there are faster algorithms, the time that it would have taken to implement them would not have been worth it for the minimal speed increases that they would have given. Recently the practice of using a favouring function (λ) to increase the chance of a split where one of the sides has no objects has become quite common showing consisent speed increases.

$$\lambda(P_L, P_R, N_L, N_R) = \begin{cases} 80\% & (N_L \equiv 0 \vee N_R \equiv 0) \wedge (P_L \neq 1 \wedge P_R \neq 1) \\ 100\% & \text{otherwise} \end{cases}$$

$$C(P_L, P_R, N_L, N_R) = \lambda(P_L, P_R, N_L, N_R)(K_T + K_I(P_L * N_L + P_R * N_R))$$

Algorithm 1: Surface Area Heuristic.

```
(Cost, Side)  SAH(p, V, N_L, N_R, N_P)
{
    Voxel  V_L, V_R;
    voxel_split(p, V, &V_L, &V_R);
    P_L  =  SA(V_L) / SA(V);
    P_R  =  SA(V_R) / SA(V);
    C_L  =  C(P_L, P_R, N_L + N_P, N_R);
    C_R  =  C(P_L, P_R, N_L, N_R + N_P);
    return  min((C_L, LEFT), (C_R, RIGHT));
}
```

Algorithm 2: Classify.

```
(T_R,T_L) classify(tree,T,p,N_L,N_R,N_P)
{
  T_L = null; T_R = null;
  T_L = malloc(N_L+N_P);
  T_R = malloc(N_R+N_P);
  i_TL = i_TR = 0
  for(t < T)
  {
    is_left = is_right = false
    for(j = 0; j < 3; j++)
    {
      t_b = t_{j,p_k};

      if(t_b < p_b)
        is_left = true;

      if(t_b > p_b)
        is_right = true;
    }

    if(¬is_left ∧ ¬is_right)  // planar
    {
      if(p_Side ≡ RIGHT)
        T_R[i_TR++] = t;
      else
        T_L[i_TL++] = t;
    }
    if(is_left)
      T_L[i_TL++] = t;
    if(is_right)
      T_R[i_TR++] = t;

  }
  return (T_R,T_L);
}
```

Algorithm 3: Find Split Plane. O(n log n)

```
(N_L,N_R,N_P,p,side) find_plane(tree,V,T)
{
  best_cost = ∞; best_p = null; side = null; E = null;
  best_N_L = best_N_R = best_N_P = 0;

  j = 0;
  for(k = 0; k < tree->k; k++)
  {
    E = malloc(2|T|);
    for(t in T)
    {
      Voxel B;
      voxel_gen_from_tri(&B, t);
      voxel_clip(&B, V);
      if(voxel_is_planar(B, k))
      {
        E[j++] = {t, B_min,k, k, PLANAR};
      }
      else
      {
        E[j++] = {t, B_min,k, k, START};
        E[j++] = {t, B_max,k, k, END};
      }
    }
    sort(E); // explain later
    N_L = N_P = 0;
    N_P = |E|;
    for(i = 0; i < |E|)
    {
      p = E[i];
      P_START = P_END = P_PLANAR = 0;

      while(i < |E| ∧ E[i]_b ≡ p_b ∧ E_type ≡ END)
      {P_END++; i++;}

      while(i < |E| ∧ E[i]_b ≡ p_b ∧ E_type ≡ PLANAR)
      {P_PLANAR++; i++;}

      while(i < |E| ∧ E[i]_b ≡ p_b ∧ E_type ≡ START)
      {P_START++; i++;}

      N_P = P_PLANAR; N_R -= P_PLANAR; N_R -= P_END;

      sah_data = SAH(k, p_b, V, N_L, N_R, N_P);

      if(sah_data_Cost < best_cost)
      {
        best_cost = sah_data_Cost;
        best_p = p;
        best_side = sah_data_Side;
        best_N_L = N_L; best_N_R = N_R; best_N_P = N_P;
      }
      N_L += P_PLANAR; N_L += P_START; N_P = 0;
    }
  }
  return (best_N_L,best_N_R,best_N_P,best_p,best_side);
}
```

Algorithm 4:

```
Node gen_node(tree,V,T,depth)
{
  Node n;

  (N_L,N_R,N_P,p,side) = find_plane(tree,V,T);
  n_p = p;

  if(depth ≡ tree_max_depth ∨ p_cost > K_I|T|)
  {
    n_T = T;
    return n;
  }

  Voxel V_L,V_R;
  voxel_split(p,V, &V_L, &V_R);
  (T_R,T_L) = classify(tree,T,p,N_L,N_R,N_P);
  n_left = ge_node(tree,V_L,T_L,depth+1);
  n_right = ge_node(tree,V_R,T_R,depth+1);
  return n;
}
```

Persistent Threading

Persistent threading is a relatively new algorithm that takes advantage of a rather old strategy. It takes advantage of the improved performance of Warp synchronous execution (a warp is Nvidia's unit for a group of processors that run using SIMT, also known as a Wavefront on AMD hardware). Persistent threading enforces that only a single task (thread) is assigned to each core of the GPU. This allows for improved SIMT performance between cores of a warp/wavefront as it enforces equal distribution of work tasks (especially when there is a non-trivial workload). It gets around the limited workloads of warp synchronous programming by implementing a global work queue that each thread pulls off of. Another issue that persistent threading alleviates is unbalanced workload distribution that can happen with algorithms that don't fit the SIMD model well (such as tree traversal). The GPU scheduler can sometimes distribute work incorrectly where certain warps will be doing all of the heavy lifting for an operation, but because each core only has one thread, persistent threading can bypass the schedulers as the workload must be evenly distributed. This can greatly improve the performance of these algorithms as it takes advantage of the entire GPU.

Short stack k-d tree traversal.

This is one of the fastest methods of k-d tree traversal. It takes advantage of both the Persistent Threading algorithm and the Short Stack k-d tree traversal algorithm. The short stack k-d tree traversal algorithm is a method for traversal that uses the best parts of k-d restart (with pushdown modification) and traversal algorithms with stacks by using a bounded stack. It will keep descending the tree (and pushing down the root) until the traversal algorithm finds a split where both nodes intersect the ray. The algorithm with then disables push down and pushes the farther node to the stack. It will then continue traversing down the closest node. If no intersection is found along the closest path it will then pop the first split off the stack (the next closest to the origin) and traverse. If the stack becomes full, and then doesn't hit anything while it is being emptied, the algorithm will resort to use k-d restart with the aforementioned pushdown modification where it will then restart right before the first split.

Algorithm 5: Persistent Short Stack k-d Tree Traversal.

```
void update_state(tree_buffer, index, *type, *node, *leaf)
{
  *type = *(uint8*)(tree_buffer+index); // first value of both node and leaf is the type
  if(*type == LEAF)
  {
    kd_serialized_leaf sleaf = *(kd_leaf*)(tree_buffer + index); //need to use serialized types for this
    leaf->type = *type;
    leaf->tri_n = sleaf.tri_n; //number of triangles
    leaf->tri_offset = index+sizeof(kd_serialized_leaf);
  }
  else //NODE
  {
    *node = *(kd_node*)(tree_buffer + index);
  }
}

(index,t,u,v) traverse(ray_buffer, indices, vertices, tree_buffer)
{
  blocksize_x = STREAM_PROCESSORS_PER_SIMT_GROUP;
  blocksize_y = SIMT_GROUPS_PER_STREAM_MULTIPROCESSOR;

  x = SM_ID % blocksize_x; //Id within the SIMT GROUP
  y = SM_ID / blocksize_x; //Id of the SIMT GROUP within the Stream Multiprocessor

  //NOTE: shared memory is called local memory in OpenCL
  shared volatile next_ray_array[blocksize_y]; //shared across all processors in the multiprocessor
  shared volatile ray_count_array[blocksize_y];

  //NOTE: In the implementation, the warp_counter is initialised on the cpu and copied to the GPU.
  global volatile warp_counter; //global memory is shared accross the entire device.

  next_ray_array[y] = 0;
  ray_count_array[y] = 0;

  (node_ptr, min, max) stack[STACK_SIZE];
  stack_pointer = 0;

  ray r;
  t_hit = ∞;
  hit_info = [0 0]^T;
  tringle_index = 0;
  t_min = t_max = 0;
  scene_min = 0; scene_max = ∞;
  kdtree_node root, node;
  kdtree_leaf leaf;
  current_type = NODE;
  pushdown = false;
  ray_index = 0;

  while(true)
  {
    shared volatile int* local_pool_ray_count = ray_count_array+y; //get this SIMT groups ray count
    shared volatile int* local_pool_next_ray  = next_ray_array+y; //get this SIMT groups next ray

    if(x ≡ 0 ∧ *local_pool_ray_count ≤ 0)
    {
      *local_pool_next_ray = atomic_add(warp_counter, BATCH_SIZE); //retrieve and incriment

      *local_pool_ray_count = BATCH_SIZE;
    }

    ray_index = *local_pool_next_ray + x;
    if(ray_index ≥ |ray_buffer|)
      break;

    if(x ≡ 0)
    {
      *local_pool_next_ray += 32;
      *local_pool_ray_count -= 32;
    }

    r = ray_buffer[ray_index];
    t_hit = ∞

    if(!collides_voxel(SCENE_V, r, &scene_min, &scene_max))
    {
      scene_max = ∞
    }

    stack_pointer = 0;
    root = *tree_buffer;

    while(t_max < scene_max)
    {
      if(stack_pointer ≡ 0)
      {
        node = root;
        current_type = NODE;
        t_min = t_max;
        t_max = scene_max;
        pushdown = true;
      }
      else //pop a node off the stack
      {
        stack_pointer--;
        svalue = stack[stack_pointer];
        t_min = svalue_min;
        t_max = svalue_max;
        update_state(tree_buffer, svalue_node_pointer, &current_type, &node, &leaf);
        pushdown = false;
      }

      while(current_type ≠ LEAF)
      {
        t_split = (node_b - r_origin,k) / r_dir,k;

        left_is_close = (r_orig,k < node_b ∨ (r_orig,k ≡ node_b ∧ r_dir,k ≤ 0));

        first = left_is_close ? node_left : node_right;
        second = left_is_close ? node_right : node_left;

        if(t_split > t_max ∨ t_split ≤ 0)
        {
          update_state(tree_buffer, first, &current_type, &node, &leaf);
        }
        else if(t_split < t_min)
        {
          update_state(tree_buffer, second, &current_type, &node, &leaf);
        }
        else
        {
          stack[stack_pointer++] = {second, t_split, t_max}; //push
          update_state(tree_buffer, first, &current_type, &node, &leaf);
          t_max = t_split;
          pushdown = false;
        }

        if(pushdown)
        {
          root = node;
        }
      }

      for(t = 0; t,|leaf_num_triangles|; t++)
      {
        vec3 tri[3];
        offset = *(uint*)(kd_tree+leaf_triangle_start+(t*sizeof(uint)));

        for(j = 0; j < 3; j++) //read triangle indices
          tri[j] = read_texture(vertices, read_texture(indices, offset+j)_x)_xyz;

        hit_coords = [0 0 0]^T;
        if(collides_triangle(tri, &hit_coords, r))
        {
          if(hit_coords_t ≤ 0)
            continue;
          if(hit_coords_t < t_hit)
          {
            t_hit = hit_coords_t;
            hit_info = hit_coords_uv;
            tri_index = offset;
          }
        }
      }
    }

    result = {0};
    if(t_hit ≠ ∞)
    {
      result = {tri_indx, t_hit, hit_info_u, hit_info_v,}
    }

    return result;
  }
}
```