THIS IS THE TITLE

Favouring:

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

Cost:

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

SAH:

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: 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;

  for(k = 0; k < tree->k; k++)
  {
    j = 0;
    E = malloc(2|T|);
    for(t in T)
    {
      Voxel B;
      B = voxel_gen_from_tri(t);
      B = 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); //sort the events by b
    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 .3: 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 .4: GenerateTree.

```
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  =  gen_node(tree, V_L, T_L, depth+1);
  n_right  =  gen_node(tree, V_R, T_R, depth+1);
  return  n;
}
```

Algorithm .5: Persistent Short Stack K-D Tree Traversal.

```
(type,node,leaf) update_state(tree_buffer,i)
{
  type = tree_buffer_{i,type};
  leaf = node = null;
  if(type == LEAF)
  {
    leaf = tree_buffer_{i,leaf};
  }
  else //NODE
  {
    node = tree_buffer_{i,node};
  }

  return {type,node,leaf};
}

(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.
  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];

  ray r;
  hit = [0 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)
  {
    //get this SIMT groups ray count
    shared volatile int* local_pool_ray_count = ray_count_array+y;
    //get this SIMT groups next ray
    shared volatile int* local_pool_next_ray  = next_ray_array+y;

    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 = ∞
    (did_hit, scene_min, scene_max) = collides_voxel(SCENE_V, r);
    if (!did_hit)
        scene_max = ∞;

    stack.clear();
    root = tree_buffer_0;

    while (t_max < scene_max)
    {
        if (|stack| ≡ 0)
        {
            node = root;
            current_type = NODE;
            t_min = t_max;
            t_max = scene_max;
            pushdown = true;
        }
        else //pop a node off the stack
        {
            (index, t_min, t_max) = stack.pop();
            (type,node,leaf) = update_state(tree_buffer, index);
            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)
                (type,node,leaf) = update_state(tree_buffer, first);
            else if (t_split < t_min)
                (type,node,leaf) = update_state(tree_buffer, second);
            else
            {
                stack.push({second, t_split, t_max});
                (type,node,leaf) = update_state(tree_buffer, first);
                t_max = t_split;
                pushdown = false;
            }
            if(pushdown)
                root = node;
        }

        for(t = 0; t,|leaf_num_triangles|; t++)
        {
            vec3 tri[3];
            offset = tree_buffer_leaf_triangle_offset;

            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 < hit_t)
                {
                    hit = hit_coords;
                    tri_index = offset;
                }
            }
        }
    }
```

```
        result = {0};
        if (hit_t ≠ ∞)
        {
          result = {tri_indx, hit_t, hit_u, hit_v,};
        }

        return result;
    }
}
```