Lua, k-d Trees and Boids

Nathan Cournia

acnatha@vr.clemson.edu

Clemson University

Motivation

- Create a simple testbed app. to experiment with the following:
  - DLL loading
  - Exceptions
  - Inheritance Chains
  - Scripting
    - Perform all world/entity logic
    - Trigger events which call scripts

Motivation (cont.)

- Motivation Continued
  - Scene Graphs
    - Have thousands of moving objects
    - 100,000+ triangles
    - Per-pixel lighting
    - Support new hardware features (fragment programs, VBO, etc.)
  - Truly make everything data driven
- In the end have a series of objects that can be plugged into existing code

Monster Testbed App. Activate

- But what will the app. actually do?
- Answer: Flocking
- But:
  - Where’s the violence?
  - Where’s the blood spray?
- New answer: Flocking with weapons

Master Plan = Violent Birds

- Create a non-interactive environment with simple world/rules
- User provides scripts which allow the birds to think
- User's `think( )` script function is given the following:
  - Bird information (position, velocity, etc.)
  - List of friends in visual range
  - List of enemies in visual range

Master Plan (cont.)

- User's `think( )` must return the following:
  - Heading in which to move
  - Desired speed
  - If the birds weapon should fire
- User shouldn't have to worry about doing complicated physics calculations or detecting collisions

Scripting Goals

- Scripts can directly modify world state
- Restrict some scripts to manipulating only parts of the world (avoid cheating)
- Scripts syntax must be easy to learn/use
- Scripts must have low overhead
- i.e. don’t use TCL

Lua

- "Moon" in Portuguese.
- Powerful light-weight programming language designed for extending applications
- Can also be used as a stand-alone language
- About 6000 lines of C!
- Grammar fits on less than a page!

Lua Features

- Dynamically typed
- Interpreted from bytecodes
- Automatic memory management (which the programmer can control)
- Procedural language

Lua Types

- Dynamically typed
- Only values have types (variables don't)
- Types:
  - nil
  - numbers (doubles be default)
  - strings
  - functions
  - userdata (provided by host)
  - tables

Lua Tables

- Associative Arrays
- Can be treated like arrays
- Can be indexed with any value (even other tables!)
- Table values can be any value (even functions!)
- Leads to methods for object oriented programming

Lua's C Interface

- About 30 functions
- Host can:
  - Read/Write variable in Lua
  - Call Lua functions
- Lua can call registered host functions
- Host communicates with Lua via a stack

Lua's Conclusion

- Easy to learn (not Lisp!)
- Fast (20 times faster than TCL)
- 20 times slower than C
- Go learn Lua

Glue

- Lua was designed to work with C
- How do we get Lua and C++ to work together?
- We want to call C++ object methods from Lua
- We can't get the address of a method in an object
- We can get the address of a static method in an object

Glue Example

```
class lua_script {
 public:
  lua_script( void );
  bool load( const std::string& filename );
  bool run( entity *ent, const std::string& method );
  bool add_function( const std::string& func, lua_CFunction f );
  void close( void );

 private:
  static int set_velocity( lua_State *vm );
  static int get_position( lua_State *vm );

  lua_State *m_vm;
};
```

Glue Example (cont.)

```
int lua_script::get_position( lua_State *vm ) {
  if( lua_gettop( vm ) != 1 ) {
    std::cerr << "error: getPosition( id )" << std::endl;
    return 0;
  }

  int id = static_cast<int>( lua_tonumber( vm, 1 ) );
  entity *ent = world.get_entity( id );
  if( !ent ) return 0;
  lua_pushnumber( vm, ent->get_current( )->position.x( ) );
  lua_pushnumber( vm, ent->get_current( )->position.y( ) );
  lua_pushnumber( vm, ent->get_current( )->position.z( ) );
  return 3;
}
```

Let’s Be Friends

- User is able to get a list of friends/enemies within a certain range
- 1000’s of moving objects in the scene
- Nearest neighbor problem
- Problem: Checking if each object is in range will be to slow
- Solution: Use a spatial subdivision data structure to help find neighbors
- Once we have neighbors, sort them into friend/foe lists

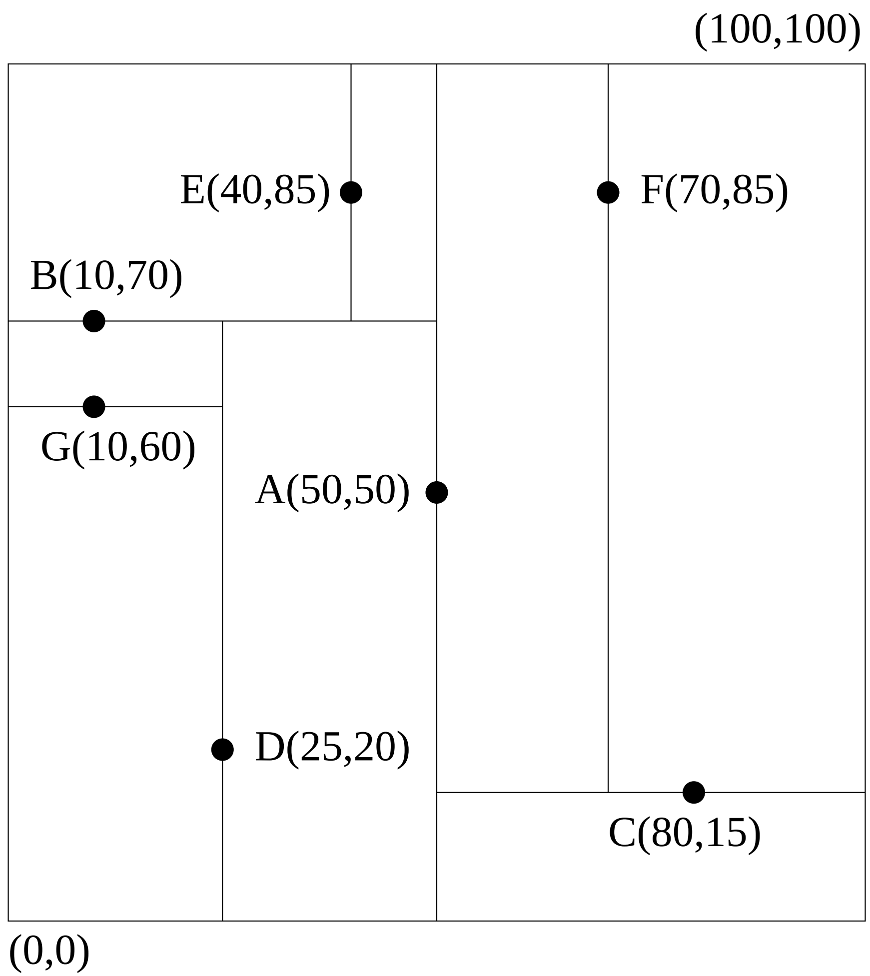
k-d Tree

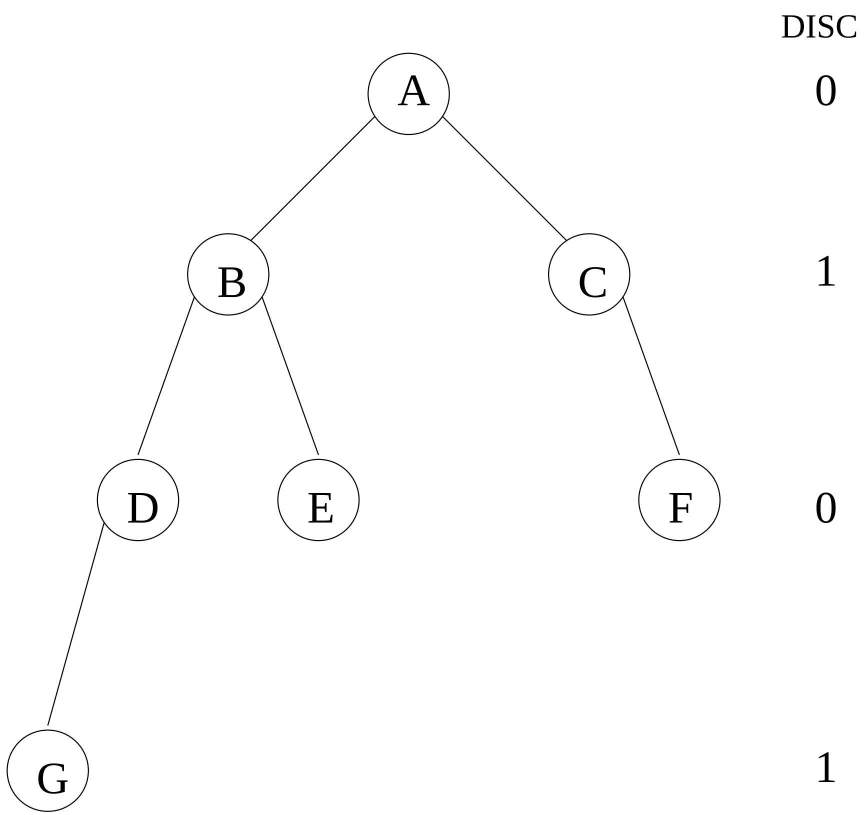
- Multidimensional binary tree
- k is the dimensionality of the search space
- Complexities:
  - Insert: $O(\log n)$
  - Delete: $O(\log n)$
  - Optimization: $O(n \log n)$
  - Search optimized: $O(\log n)$
- Search in an unoptimized tree usually visits $1.386 \log_2 n$ nodes

k-d Tree Discriminator

- Associated with each node is a discriminator $[0, k-1]$
- All nodes on any given level of the tree have the same discriminator
- For any node P, let j be $\mathrm{DISC}(P)$
  - Then for any node Q in $\mathrm{LEFT}(P)$, $K_j(Q) < K_j(P)$
  - Then for any node R in $\mathrm{RIGHT}(P)$, $K_j(R) > K_j(P)$

k-d Tree Example (k = 2)





k-d Tree Insertion/Search

```
void kdtree<D,K,T,C>::insert( kdtree<D, K, T, C>::node*& tree,
                              T& data, size_t disc ) {
  if( !tree ) {
    tree = do_insert( data, disc, NULL, NULL );
    return;
  }

  int suc = m_compare( data, tree->data, disc );
  if( suc < 0 ) {
    insert( tree->left, data, next_disc( disc ) );
  } else if( suc > 0 ) {
    insert( tree->right, data, next_disc( disc ) );
  }
}
```

k-d Tree Nearest Neighbor

```
void kdtree<D,K,T,C>::neighbors( kdtree<D,K,T,C>::node* tree,
                                 std::list<T>& results, T& data,
                                 K distance ) {
  if( !tree ) return;
  K delta = m_compare.diff( data, tree->data, tree->disc );
  if( delta < 0 ) {
    neighbors( tree->left, results, data, distance );
    if( (delta * delta) < distance )
      neighbors( tree->right, results, data, distance );
  } else {
    neighbors( tree->right, results, data, distance );
    if( (delta * delta) < distance )
      neighbors( tree->left, results, data, distance );
  }
```

k-d Tree Nearest Neighbor (cont.)

```
    delta = m_compare.diff( data, tree->data );
    if( (delta * delta) < distance ) {
      results.push_back( tree->data );
    }
  }
```

k-d Tree C++ Tangent

- Creating a generic k-d tree in C++ is simple
- We must be able to determine if a node is less than or greater than another node given a discriminate
- Use a templates and functors

k-d Tree Functor Example

```
kdtree<size_t DIMS, class DT, class T, class SUCCESSOR>

struct vector3_successor {
  int operator()( const math::vector3* lhs,
                  const math::vector3* rhs, size_t dim ) {
    assert( dim < 3 );
    const math::vector3 &a = *lhs, &b = *rhs;
    for( unsigned int i = 0; i < 3; ++i ) {
      unsigned int j = (i + dim) % 3;
      if( a( j ) < b( j ) ) return -1;
      if( a( j ) > b( j ) ) return 1;
    }
    return 0;
  }
};
```

k-d Tree Example (cont.)

- Isn't this slow? No!
- Faster than the C equivalent: function pointers (qsort)
- Why? Compilers can optimize the code in the functor
- Over 600% faster!

Flocking

- Simulates the behavior of a group (herd, school, swarm, etc.)
- Made up of individual autonomous agents called boids
- Can be though of as a specialized particle system
- Stateless algorithm

Flocking Rules

- Algorithm is marked by four rules (steering behaviors)
  - Separation - Avoid crowding
  - Alignment - Move in the same direction local flock mates are moving
  - Cohesion - Move towards the center of the flock's mass
  - Avoidance - Avoid obstacles, flock mates, enemies
- Emergent Behavior

Flocking Demo

Future/Conclusions/Questions

- Most of the things mentioned on the first slide are not done
- Is there a better way to compute the nearest neighbor?
- Running scripts through Lua is cheap, but not free
  - Add script scheduler.
- Use OO in Lua.