Generational Index

Intent

Allow an entity to query it's components without sacrificing performance

Motivation

One method of handling entities in game design is using an entity component system (ECS). Using composition, entities can be comprised of components rather than dumping all functionality into one massive class. This makes updating game state more efficient. For example, the physics engine now can process just the physics components it needs rather than having to load the entire entity. This does create new issues, like what data structure should we use to store all these components?

Array

```
type Entity struct {
  PhysicsId   int
  AnimationId int
  StateId     int
}

physics := []PhysicsComponents{}
animations := []AnimationComponents{}
states := []StateComponents{}
```

Arrays or vectors aren't a terrible choice. Indexing and inserting at the end of the array are very fast operations. Unfortunately, deleting and maintaining order is a lot more complicated. For starters each index is a reference for the entity to find a component. When an entity is deleted, we have the choice of nulling the component and risking the array growing out of control or being forced to delete each index in order from all component arrays. This is even more complex if an entity is not guaranteed to have all components and we potentially open ourselves to an entity referencing the wrong component.

Benchmark @ 1,000,000 entities	**Executions (1 second)**	**Average time per execution**
BenchmarkSliceIndexing-16	105072446	10.59 ns/op
BenchmarkSliceInsert-16	60619332	19.04 ns/op
BenchmarkSliceDelete-16	2140	535028 ns/op

Hash map

```
type Entity struct {
  PhysicsId   int
  AnimationId int
  StateId     int
}

physics := map[int]PhysicsComponents{}
animations := map[int]AnimationComponents{}
states := map[int]StateComponents{}
```

Hash maps are a very flexible data structure. One key advantage over arrays is the ability to determine if a key has been removed through some implementation of a has_key? method. Inserting, deleting, and indexing are pretty fast, but not as fast as the array and performance suffers as the hash map grows.

Benchmark @ 1,000,000 entities	Executions (1 second)	Average time per execution
BenchmarkHashmapIndexing-16	10062358	107.4 ns/op
BenchmarkHashmapInsert-16	6372176	166.3 ns/op
BenchmarkHashmapDelete-16	1000000000	0.7348 ns/op

This could be the result of the implementation of the underlying hashing function or because of more frequent cache misses. For reference, this is data taken from the Google SRE Workbook concerning common latency numbers:

Operation	Time (ns)
L1 cache reference	1
L2 cache reference	4
Main memory reference	100

Ideally, we would want some data structure that still has the fast indexing and inserting of the array, but also can determine if an index is no longer active to avoid the expensive delete process.

Generational Index

The generational index is a common strategy in game development that aims to solve this problem. At it's core it is an array with some other added features.

```go
type GenerationalIndex struct {
  index      int
  generation int
}

type allocatorEntry struct {
  isLive     bool
  generation int
}

type GenerationalIndexAllocator struct {
  entries []allocatorEntry
  free    []int
}
func (g *GenerationalIndexAllocator) Allocate() GenerationalIndex
func (g *GenerationalIndexAllocator) Deallocate(i GenerationalIndex)
func (g *GenerationalIndexAllocator) IsLive(i GenerationalIndex) bool
```

To break this down, the underlying array is `entries`. When creating a new generational index, the Allocator will check to see if any indices are available in `free`. If `free` is empty, the `entries` array grows by 1 and a new `GenerationtionalIndex` is returned with an `index: len(entries)-1` and a `generation: 1`. If `free` is not empty, that means that `entries` has indices that have been Deallocated and were pushed into `free`. To re-use these, we pop an index off of `free`, increment `generation` at that index in `entries` by 1, and create a new GenerationalIndex with this `index` and `generation`.

This does a few neat things. First it means that `entries` will only ever be as big as the highest number of active components. In addition, `isLive` and `free` offer convenient options for knowing if an index was de-referenced. Finally, `generation` is a final safeguard that allows for indices to be re-used without the worry of some entity still referencing it. If there is a mismatch in the `GenerationalIndex.generation` and the `allocatorEntry.generation` then that index is considered stale and unsafe to use.

Benchmark @ 1,000,000 entities	Executions (1 second)	Average time per execution
BenchmarkGenerationalIndexIndexing-16	45370582	22.45 ns/op
BenchmarkGenerationalIndexInsert-16	32015978	44.20 ns/op
BenchmarkGenerationalIndexDelete-16	1000000000	0.8059 ns/op

This pattern still performs well at scale for inserting and index. For deleting, it is on par with the hash map. The primary trade-off with this technique is complexity and memory. My implementation of the GenerationalIndex in go is just shy of 200 lines of code and has the added overhead of requiring 3 arrays to support it. It also requires that all operations must go through the `GenerationalIndexAllocator` which can be cumbersome at times.