Arcade - Technical Implementation Document

Core Architecture

Dynamic Library Loading
The core system uses a template-based dynamic library loader (`DLLoader`) that handles:
- Runtime loading of shared libraries
- Symbol resolution
- Error handling
- Resource cleanup

```cpp
template <typename T>
class DLLoader {
void* handle;
std::shared_ptr<T> instance;
// ...
};
```

Event System
Events are handled through an enumeration system that provides a uniform interface across all display libraries:

```cpp
enum class EventType {
NONE, QUIT, MOVE_UP, MOVE_DOWN, MOVE_LEFT, MOVE_RIGHT,
ACTION, PAUSE, MENU, NEXT_LIB, PREV_LIB, NEXT_GAME, PREV_GAME
};
```

Game State Management
Games implement a state machine pattern:

```cpp
enum class GameState {
MENU, PLAYING, PAUSED, GAME_OVER, WIN
};
```

Graphics Libraries Implementation

Common Interface
All graphics libraries implement the `IGraphical` interface:

```cpp
class IGraphical {
```

```
virtual void init(int width, int height, const std::string &title) = 0;
virtual void close() = 0;
virtual bool isOpen() const = 0;
virtual void clear() = 0;
virtual void display() = 0;
// ...
};
```

Graphics Primitives
Basic shapes and text rendering capabilities:
- Rectangles
- Circles
- Sprites
- Text with color and position

Game Implementations

Snake Game
- Grid-based movement system
- Collision detection with walls and self
- Growing mechanism on food collection
- Score tracking

Pacman Game
- Ghost AI with basic pathfinding
- Dot collection mechanics
- Wall collision system
- Multiple ghost behaviors

Nibbler Game
- Snake-like mechanics with walls
- Level progression system
- Increasing difficulty
- Map loading system

Map Loading System

File Format
Maps are stored in text files with the following format:
```
####################
#........##........#
#.####...##...####.#
// ...
```

Loading Mechanism

```cpp
class MapLoader {
static std::vector<std::string> loadMap(const std::string& filename);
static std::string getDefaultMap(const std::string& gameName);
};
```

Memory Management

RAII Principles
- Smart pointers for resource management
- Exception-safe design
- Proper cleanup in destructors

Resource Handling

```cpp
std::unique_ptr<DLLoader<IGraphical>> _graphicalLoader;
std::vector<std::shared_ptr<IGraphical>> _previousGraphicals;
std::unique_ptr<DLLoader<IGame>> _gameLoader;
```

Performance Considerations

Frame Timing
- Delta time-based updates
- Frame rate limiting
- Vsync support in graphics libraries

Resource Loading
- Lazy loading of assets
- Cache management
- Error recovery systems

Testing and Debugging

Debug Features
- State logging
- Error reporting
- Performance monitoring

Error Handling
- Exception hierarchy
- Safe state recovery
- Graceful degradation

Future Improvements

Potential Enhancements
1. Additional games
2. More graphics libraries
3. Network multiplayer
4. Improved AI systems
5. Enhanced graphics effects