Documentation Développeur pour le Projet Codec

Ce document détaille l'architecture du projet, les fichiers principaux, leurs responsabilités, et les choix techniques effectués. Il fournit également des informations sur l'implémentation et la raison d'être des fonctionnalités.

Structure Générale

Le projet est organisé en plusieurs fichiers source et en-tête, regroupés dans des modules fonctionnels. Chaque module est responsable d'une partie spécifique de l'application.

Arborescence des Fichiers

```
└── include
    ├── cli
    │   └── cli.h
    ├── codec
    │   ├── codec.h
    │   ├── compression.h
    │   └── decompression.h
    ├── common
    │   └── common.h
    ├── config
    │   └── config.h
    ├── core
    │   └── quadtree.h
    ├── grid
    │   └── segmentation_grid.h
    ├── io
    │   └── pgm.h
    ├── logger
    │   ├── logger_private.h
    │   └── logger.h
└── src
    ├── cli
    │   └── cli.c
```

```
    ├── codec
    │   ├── codec.c
    │   ├── compression.c
    │   └── decompression.c
    ├── common
    │   └── common.c
    ├── config
    │   └── config.c
    ├── core
    │   └── quadtree.c
    ├── grid
    │   └── segmentation_grid.c
    ├── io
    │   └── pgm.c
    ├── logger
    │   └── logger_utils.c
    └── main.c
```

---

1. Module `config`

Fichiers

- `config.h` : Définit les constantes, structures, et prototypes des fonctions liées à la configuration.
- `config.c` : Implémente la logique pour initialiser et manipuler les configurations.

Détails du Fichier `config.h`

Rôle:

- Fournir un espace centralisé pour les paramètres du programme.
- Déclarer la structure de configuration et les constantes associées.

Contenu:

1. **Constantes**:
   - Définit des valeurs par défaut pour les fichiers de sortie et le paramètre `alpha`.

```
#define DEFAULT_COMPRESS_OUTPUT "default_compress_output.qtc"
#define DEFAULT_DECOMPRESS_OUTPUT "default_compress_input.pgm"
#define DEFAULT_ALPHA 1.0f
```

2. **Structure** `config_t` :
   - Centralise tous les paramètres de configuration.
   - Inclut des champs booléens pour indiquer si le programme doit compresser, décompresser ou générer une grille.
   - Ajoute des pointeurs vers les chemins de fichiers d'entrée, de sortie, et de grille.

```
typedef struct {
    bool compress;           // Compression activée ?
    bool decompress;         // Décompression activée ?
    bool generate_grid;      // Générer une grille ?
    const char *input_file;  // Chemin du fichier d'entrée
    const char *output_file; // Chemin du fichier de sortie
    const char *grid_file;   // Chemin pour la grille
    float alpha;             // Paramètre de compression
} config_t;
```

3. **Fonctions**:
   - `void config_init(config_t *config);` : Initialise les champs de la structure `config_t` avec des valeurs par défaut.

Détails du Fichier `config.c`

Rôle:

- Implémenter la logique pour initialiser la structure `config_t`.

Fonctionnalité:

1. **Initialisation des Valeurs**:
   - La fonction `config_init` met à zéro la mémoire de la structure `config_t` à l'aide de `memset`.
   - Définit la valeur par défaut pour `alpha` (compression à 1.0 par défaut).

```
void config_init(config_t *config) {
    memset(config, 0, sizeof(config_t));
```

```
    config->alpha = DEFAULT_ALPHA;
}
```

Choix Techniques:

- **Utilisation de `memset`**:
  - Permet de garantir que tous les champs de la structure sont initialisés à `0` ou `NULL`.
- **Séparation entre déclaration et implémentation**:
  - Améliore la lisibilité et la maintenabilité du code.
  - Permet de centraliser les valeurs par défaut dans `config.h`.

---

2. Module `cli`

Fichiers

- `cli.h`: Déclare les fonctions pour l'analyse des arguments en ligne de commande.
- `cli.c`: Implémente la logique pour interpréter et valider les options CLI.

Détails du Fichier `cli.h`

Rôle:

- Fournir une interface permettant de capturer et valider les arguments en ligne de commande.
- Faciliter le passage des options à d'autres modules via la structure `config_t`.

Contenu:

1. **Fonctions Principales**:
   - `void cli_print_help(void);` : Affiche l'aide sur les options disponibles.
   - `bool cli_parse_arguments(int argc, char **argv, config_t *config);` : Analyse et valide les arguments fournis.

Détails du Fichier `cli.c`

Rôle:

- Interpréter les arguments fournis par l'utilisateur via la ligne de commande.
- Valider les options et signaler les erreurs d'utilisation.

Fonctionnalité:

1. **Affichage de l'Aide**:
   - La fonction `cli_print_help` affiche un guide d'utilisation détaillé des options disponibles :

```
void cli_print_help(void) {
    printf("Usage: codec [options]\n"
           "Options:\n"
           "  -c              Compress the input PGM file\n"
           "  -u              Decompress the input QTC file\n"
           "  -i <input>      Input file path\n"
           "  -o <output>     Output file path\n"
           "  -g              Generate segmentation grid\n"
           "  -a <alpha>      Compression parameter (default: 1.0)\n"
           "  -h              Display this help\n");
}
```

2. **Analyse des Options**:
   - `handle_option_with_argument` :
     - Gère les options comme `-i`, `-o`, ou `-a` qui requièrent un argument.

```
static bool handle_option_with_argument(const char opt, int *i, int
argc, char **argv, config_t *config) {
    if (++(*i) >= argc) {
        fprintf(stderr, "Error: Missing argument for -%c\n", opt);
        return false;
    }
    switch (opt) {
        case 'i': config->input_file = argv[*i]; break;
        case 'o': config->output_file = argv[*i]; break;
        case 'a': {
            float alpha = atof(argv[*i]);
            if (alpha <= 0) {
                fprintf(stderr, "Error: Invalid alpha value\n");
                return false;
            }
            config->alpha = alpha;
```

```
            break;
        }
        default:
            fprintf(stderr, "Error: Invalid option '-%c'\n", opt);
            return false;
    }
    return true;
}
```

- `handle_flag_option` :
  - Gère les options simples comme `-c` , `-u` , et `-h` .

```
static bool handle_flag_option(const char opt, config_t *config) {
    switch (opt) {
        case 'c': config->compress = true; break;
        case 'u': config->decompress = true; break;
        case 'h': cli_print_help(); exit(EXIT_SUCCESS);
        default:
            fprintf(stderr, "Error: Unknown option '-%c'\n", opt);
            return false;
    }
    return true;
}
```

3. **Validation des Options**:
   - Vérifie que les combinaisons d'options sont valides.
   - Par exemple, interdit de spécifier `-c` et `-u` simultanément.

```
static bool validate_config(config_t *config) {
    if (config->compress && config->decompress) {
        fprintf(stderr, "Error: Cannot specify both compression and
decompression\n");
        return false;
    }
    if (!config->compress && !config->decompress) {
        fprintf(stderr, "Error: Must specify either compression or
decompression\n");
        return false;
    }
    if (!config->input_file) {
        fprintf(stderr, "Error: Input file not specified\n");
        return false;
    }
    if (!config->output_file) {
```

```
        config->output_file = config->compress ? DEFAULT_COMPRESS_OUTPUT :
DEFAULT_DECOMPRESS_OUTPUT;
    }
    return true;
}
```

4. **Analyse Complète**:
   - Combine toutes les étapes ci-dessus pour analyser et valider les arguments.

```
bool cli_parse_arguments(int argc, char **argv, config_t *config) {
    config_init(config);
    for (int i = 1; i < argc; i++) {
        const char *arg = argv[i];
        if (arg[0] != '-' || arg[1] == '\0') {
            fprintf(stderr, "Error: Invalid argument format: %s\n", arg);
            return false;
        }
        if (strchr("ioga", arg[1])) {
            if (!handle_option_with_argument(arg[1], arg, &i, argc, argv,
config)) {
                return false;
            }
        } else if (!handle_flag_option(arg[1], config)) {
            return false;
        }
    }
    return validate_config(config);
}
```

---

3. Module `logger`

Fichiers

- `logger.h` : Déclare l'interface publique de la bibliothèque de logging.
- `logger_private.h` : Définit les structures internes, constantes, et fonctions utilitaires privées pour le logger.
- `logger_utils.c` : Implémente toutes les fonctionnalités de la bibliothèque, y compris les utilitaires internes et les fonctions publiques.

Détails du Fichier `logger.h`

Rôle:

Fournir une interface simple et flexible pour gérer des messages de log, des barres de progression, et des statistiques détaillées dans la console.

Contenu:

1. **Types de Messages (`log_level_t`)**:
   - Permet de classifier les messages selon leur type : information, succès, avertissement ou erreur.

```
typedef enum {
    LOG_LEVEL_INFO,
    LOG_LEVEL_SUCCESS,
    LOG_LEVEL_WARN,
    LOG_LEVEL_ERROR
} log_level_t;
```

2. **Configuration du Logger (`logger_config_t`)**:
   - Permet de personnaliser l'affichage des logs (couleurs, timestamps, activation).

```
typedef struct {
    bool enabled;
    bool use_colors;
    bool show_timestamp;
} logger_config_t;
```

3. **Fonctions Principales**:
   - Configurer le logger : `void logger_configure(logger_config_t config);`
   - Gérer les messages : `void log_message(log_level_t level, const char *format, ...);`
   - Suivre la progression : `void log_progress(double percentage);`
   - Afficher des séparateurs, en-têtes, et statistiques.
4. **Macros de Commodité**:
   - Simplifient l'appel des logs selon leur type.

```
#define log_info(...) log_message(LOG_LEVEL_INFO, __VA_ARGS__)
#define log_success(...) log_message(LOG_LEVEL_SUCCESS, __VA_ARGS__)
```

```
#define log_warn(...) log_message(LOG_LEVEL_WARN, __VA_ARGS__)
#define log_error(...) log_message(LOG_LEVEL_ERROR, __VA_ARGS__)
```

Détails du Fichier `logger_private.h`

Rôle:

- Fournir des définitions internes au logger pour gérer les couleurs, les thèmes, les symboles de terminal, et la configuration globale.
- Ces structures et fonctions sont exclusivement utilisées en interne dans la bibliothèque.

Contenu:

1. **Structure pour les Couleurs (`ColorScheme`)**:
   - Permet de définir les styles visuels (texte normal, gras, dim) pour les logs.

```
typedef struct {
    const char *regular;
    const char *bold;
    const char *dim;
} ColorScheme;
```

2. **Symboles de Terminal (`TerminalSymbols`)**:
   - Gère les caractères utilisés pour les bordures, barres de progression, et autres éléments graphiques.

```
struct TerminalSymbols {
    const char *bar_full;
    const char *bar_empty;
    const char *corner_tl;
    const char *corner_tr;
    const char *corner_bl;
    const char *corner_br;
    const char *line_h;
    const char *line_v;
    const char *bullet;
};
```

3. **Configuration de Thème (`ThemeConfig`)**:

- Gère les couleurs et styles des différents éléments de texte.

```
struct ThemeConfig {
    const char *header;
    const char *border;
    const char *label;
    const char *value;
    const char *timestamp;
};
```

4. **Configuration de Mise en Page (`LayoutConfig`)**:
   - Permet de personnaliser les dimensions des éléments graphiques dans les logs.

```
struct LayoutConfig {
    uint16_t separator_width;
    uint16_t progress_width;
    uint16_t label_width;
};
```

5. **Fonctions Utilitaires Internes**:
   - `void ensure_initialized(void);` : Garantit que le logger est correctement initialisé avant utilisation.
   - `void safe_vfprintf(FILE *out, const char *format, va_list args);` : Version sécurisée de `vfprintf`.
   - `ColorScheme get_level_colors(log_level_t level);` : Retourne les couleurs associées à un niveau de log.
   - `const char *get_level_symbol(log_level_t level);` : Retourne le symbole associé à un niveau de log.
6. **Variables Globales**:
   - `LoggerState g_state;` : Stocke l'état global du logger (configurations, flux de sortie, etc.).
   - Pointeurs constants vers les structures de symboles, couleurs et thèmes.

Détails du Fichier `logger_utils.c`

Rôle:

- Implémenter toutes les fonctionnalités publiques et internes du module de logging, y compris la gestion des couleurs, des symboles, et des barres de progression.

Fonctionnalités Clés:

1. **Initialisation et Configuration**:
   - La fonction `ensure_initialized` vérifie si le logger est prêt à l'emploi et initialise les paramètres globaux si nécessaire.

```
void ensure_initialized(void) {
    if (g_state.is_initialized)
        return;
    g_state.output_stream = stdout;
#ifdef _WIN32
    system("chcp 65001 > nul"); // Activer l'UTF-8 sur Windows
#endif
    g_state.is_initialized = true;
}
```

2. **Gestion des Couleurs et Symboles**:
   - Retourne les couleurs et symboles appropriés en fonction du niveau de log.

```
ColorScheme get_level_colors(log_level_t level) {
    if (!g_state.config.use_colors) {
        return (ColorScheme){.regular = "", .bold = "", .dim = ""};
    }
    switch (level) {
        case LOG_LEVEL_INFO: return COLORS->info;
        case LOG_LEVEL_SUCCESS: return COLORS->success;
        case LOG_LEVEL_WARN: return COLORS->warning;
        case LOG_LEVEL_ERROR: return COLORS->error;
        default: return COLORS->info;
    }
}
```

3. **Affichage des Messages**:
   - Les messages de log sont formatés avec des couleurs, des symboles, et des timestamps si activés.

```
void log_message(log_level_t level, const char *format, ...) {
    ensure_initialized();
    if (!g_state.config.enabled)
        return;
```

```
    FILE *out = g_state.output_stream;
    ColorScheme colors = get_level_colors(level);

    if (g_state.config.show_timestamp) {
        print_timestamp(out);
    }

    fprintf(out, "%s%s%s ",
            colors.bold,
            get_level_symbol(level),
            COLORS->reset);

    va_list args;
    va_start(args, format);
    fprintf(out, "%s", colors.regular);
    safe_vfprintf(out, format, args);
    va_end(args);

    fprintf(out, "%s\n", COLORS->reset);
}
```

4. **Gestion de la Progression**:
    - Implémente une barre de progression dynamique affichée dans la console.

```
void log_progress(double percentage) {
    ensure_initialized();
    if (!g_state.config.enabled)
        return;

    FILE *out = g_state.output_stream;
    ColorScheme colors = get_level_colors(LOG_LEVEL_INFO);
    const int filled = (int)(LAYOUT->progress_width * percentage);
    const int empty = LAYOUT->progress_width - filled;

    fprintf(out, "\r%s%s%s",
            colors.regular,
            SYMBOLS->line_v,
            g_state.config.use_colors ? COLORS->reset : "");

    for (int i = 0; i < filled; i++) {
        fprintf(out, "%s%s%s",
                colors.bold,
                SYMBOLS->bar_full,
                COLORS->reset);
```

```
    }

    for (int i = 0; i < empty; i++) {
        fprintf(out, "%s%s%s",
                colors.dim,
                SYMBOLS->bar_empty,
                COLORS->reset);
    }

    fprintf(out, "%s%s %.1f%%%s",
            colors.regular,
            SYMBOLS->line_v,
            percentage * 100.0,
            COLORS->reset);

    fflush(out);
    g_state.is_progress_active = true;
}
```

5. **Affichage des Statistiques**:
    - Affiche les informations relatives à un fichier ou à un processus.

```
void log_file_info(const char *filename, uint32_t size, uint32_t levels,
double ratio) {
    ensure_initialized();
    if (!g_state.config.enabled)
        return;

    log_subheader("File Information");
    log_item("Name", "%s", filename);
    log_item("Dimensions", "%u\u00d7%u pixels", size, size);
    log_item("Tree depth", "%u levels", levels);

    if (ratio > 0.0) {
        const char *efficiency =
            ratio < 50.0 ? "Excellent" : ratio < 70.0 ? "Good"
                                : ratio < 85.0   ? "Fair"
                                                 : "Poor";

        log_item("Compression", "%.2f%% (%s%s%s)",
                 ratio,
                 COLORS->italic,
                 efficiency,
                 COLORS->reset);
```

```
    }
}
```

---

4. Module `common`

Rôle:

Le module `common` fournit des fonctions utilitaires génériques et des structures communes à l'ensemble du projet. Il est particulièrement utile pour les opérations de base sur les quadtree et autres calculs.

Détails du Fichier `common.h`

Rôle:

- Déclare les énumérations, fonctions utilitaires et structures globales pour les opérations courantes.

Contenu:

1. **Énumération des Quadrants (`quadrant_order_t`)**:
   - Définit l'ordre de traitement des quadrants dans une structure quadtree.

```c
typedef enum {
    QUADRANT_TOP_LEFT = 0,     /* Débuter par le coin supérieur gauche */
    QUADRANT_TOP_RIGHT = 1,    /* Ensuite le coin supérieur droit */
    QUADRANT_BOTTOM_RIGHT = 2, /* Ensuite le coin inférieur droit */
    QUADRANT_BOTTOM_LEFT = 3   /* Terminer par le coin inférieur gauche */
} quadrant_order_t;
```

2. **Tableau `quadrant_order`**:
   - Un tableau statique préconfiguré pour faciliter les traversées de quadtree selon l'ordre des quadrants.

```c
static const int quadrant_order[4] = {
    QUADRANT_TOP_LEFT,
    QUADRANT_TOP_RIGHT,
```

```
    QUADRANT_BOTTOM_RIGHT,
    QUADRANT_BOTTOM_LEFT
};
```

3. **Fonctions Utilitaires**:
   - `calculate_fourth_mean` :
     - Calcule la valeur moyenne manquante dans une division quadtree, en se basant sur les valeurs moyennes déjà connues et l'erreur de l'arrondi.

```
uint8_t calculate_fourth_mean(uint8_t parent_mean, uint8_t error,
                              uint8_t m1, uint8_t m2, uint8_t m3);
```

     - **Paramètres**:
       - `parent_mean` : Moyenne du nœud parent.
       - `error` : Erreur d'arrondi associée au calcul précédent.
       - `m1, m2, m3` : Moyennes connues des trois premiers quadrants.
     - **Retourne**: La moyenne manquante pour le quatrième quadrant.
   - `is_power_of_two` :
     - Vérifie si un entier est une puissance de deux.

```
bool is_power_of_two(uint32_t x);
```

     - **Paramètres**:
       - `x` : Entier à analyser.
     - **Retourne**: `true` si `x` est une puissance de deux, sinon `false`.

Détails du Fichier `common.c`

Rôle:

- Implémente les fonctions utilitaires déclarées dans `common.h`.

Fonctionnalités Clés:

1. `calculate_fourth_mean` :
   - Cette fonction calcule la moyenne manquante pour un quadtree en se basant sur la formule : m4=(4×parent_mean+error)−(m1+m2+m3)m_4 = (4 \times \text{parent_mean} + \text{error}) - (m_1 + m_2 + m_3)

```
uint8_t calculate_fourth_mean(uint8_t parent_mean, uint8_t error, uint8_t
m1, uint8_t m2, uint8_t m3) {
    int32_t sum = m1 + m2 + m3;
    int32_t m4 = (4 * parent_mean + error) - sum;

    return (uint8_t)m4;
}
```

- **Choix Techniques**:
  - L'utilisation de `int32_t` pour la somme et le calcul garantit que les opérations ne débordent pas, même si les entrées sont proches de la valeur maximale d'un `uint8_t`.

2. `is_power_of_two` :
   - Détermine si un entier est une puissance de deux en utilisant une opération bitwise efficace : \text{x & (x - 1)} = 0 Cela fonctionne car une puissance de deux a un seul bit à `1` dans sa représentation binaire.

```
bool is_power_of_two(uint32_t x) {
    return x && !(x & (x - 1));
}
```

- **Choix Techniques**:
  - L'utilisation de bitwise rend cette fonction très rapide et compacte.
  - La condition initiale `x &&` évite les erreurs pour `x = 0`.

Choix Techniques Généraux:

1. **Séparation des En-Têtes et de l'Implémentation**:
   - Facilite la lisibilité et la réutilisation du code.
   - Permet une compilation plus rapide lors de changements locaux.
2. **Compatibilité avec des Types Standards**:
   - Utilisation de `stdint.h` pour des types définis explicitement (comme `uint8_t` et `uint32_t`), garantissant un comportement prévisible sur toutes les plateformes.
3. **Optimisation des Performances**:
   - Les opérations bitwise et l'utilisation de types `int32_t` garantissent que les calculs sont à la fois rapides et sécurisés.

Module `quadtree`

Rôle:

Le module `quadtree` implémente les structures et fonctions nécessaires à la construction, à la manipulation et à l'analyse d'un quadtree. Il permet de décomposer une image en régions homogènes et de fournir des statistiques sur la variabilité des données.

Détails du Fichier `quadtree.h`

Rôle:

- Fournir les structures de données principales du quadtree.
- Déclarer les fonctions publiques utilisables par d'autres modules.

Contenu:

1. **Structure** `qtree_node_t`:
   - Représente un nœud dans l'arborescence avec ses propriétés essentielles :
     - Moyenne des intensités (m).
     - Erreur d'arrondi (e).
     - Uniformité de la région (u).
     - Variation locale (v).
     - Pointeurs vers ses quatre enfants (children).

```c
typedef struct qtree_node {
    uint8_t m;                      /* Moyenne d'intensité */
    uint8_t e : 2;                  /* Erreur d'arrondi */
    uint8_t u : 1;                  /* Uniformité */
    float v;                        /* Variation locale */
    struct qtree_node *children[4]; /* Sous-régions */
} qtree_node_t;
```

2. **Structure** `qtree_t`:
   - Représente l'ensemble de l'arbre quadtree :
     - Pointeur vers la racine.
     - Nombre de niveaux (n_levels).
     - Taille de l'image (doit être une puissance de 2).

```c
typedef struct {
    qtree_node_t *root; /* Racine de l'arbre */
    uint32_t n_levels;  /* Nombre de niveaux */
    uint32_t size;      /* Taille de l'image */
} qtree_t;
```

3. **Fonctions Publiques**:
   - **Initialisation et Construction**:
     - `qtree_status_t qtree_init(qtree_t *tree, uint32_t size);`
     - `qtree_status_t qtree_build(qtree_t *tree, const uint8_t *pixels, uint32_t size, const char *input_filename);`
   - **Gestion de la Mémoire**:
     - `void free_quadtree_recursive(qtree_node_t *node);`
   - **Navigation dans l'Arbre**:
     - `uint32_t qtree_parent_index(uint32_t index);`
     - `uint32_t qtree_first_child_index(uint32_t index);`
   - **Analyse des Données**:
     - `qtree_variance_stats_t calculate_variance_stats(const qtree_t *tree);`
4. **Enumération** `qtree_status_t`:
   - Indique les résultats possibles des opérations.

```c
typedef enum {
    QTREE_SUCCESS = 0,         /* Succès */
    QTREE_ERROR_MEMORY,        /* Problème de mémoire */
    QTREE_ERROR_INVALID_PARAM, /* Paramètres invalides */
    QTREE_ERROR_FORMAT         /* Problème de format */
} qtree_status_t;
```

Détails du Fichier `quadtree.c`

Rôle:

- Implémenter la logique de construction, d'analyse et de gestion mémoire des quadtrees.
- Offrir une gestion intuitive de la progression et des statistiques.

Fonctionnalités Clés:

1. **Construction de l'Arbre**:
   - **Fonction Récursive** `build_recursive`:
     - Divise l'image en quatre régions à chaque niveau jusqu'à atteindre les feuilles.
     - Calcule les propriétés de chaque nœud (moyenne, erreur, uniformité).

```
static qtree_node_t *build_recursive(const uint8_t *pixels, uint32_t
size,
                                     uint32_t level, uint32_t row,
uint32_t col,
                                     progress_tracker_t *progress);
```

   - **Suivi de la Progression**:
     - Affiche la progression en pourcentage à mesure que les nœuds sont traités.

```
progress->processed++;
if (progress->processed % (progress->total / 100) == 0 ||
    progress->processed == progress->total) {
    double percent = (double)progress->processed / (double)progress-
>total;
    log_progress(percent);
}
```

2. **Initialisation et Validation**:
   - La fonction `qtree_init` initialise les paramètres essentiels de la structure `qtree_t` et vérifie que la taille est une puissance de 2.

```
qtree_status_t qtree_init(qtree_t *tree, uint32_t size) {
    if (!tree || size == 0 || (size & (size - 1)) != 0) {
        log_message(LOG_LEVEL_ERROR, "Invalid parameters for quadtree
initialization");
        return QTREE_ERROR_INVALID_PARAM;
    }

    tree->size = size;
    tree->n_levels = (uint32_t)(floor(log2(size)));
    tree->root = NULL;

    log_message(LOG_LEVEL_INFO, "Initialized quadtree structure (%ux%u)",
size, size);
    return QTREE_SUCCESS;
}
```

3. **Analyse des Variances**:
    - Calcul des variations locales au sein de chaque nœud et génération de statistiques globales.

```
static void calculate_variances_recursive(qtree_node_t *node, float
*variances, size_t *count);
qtree_variance_stats_t calculate_variance_stats(const qtree_t *tree);
```

    - Utilisation de `qsort` pour calculer la variance médiane.

```
qsort(variances, count, sizeof(float), compare_floats);
stats.median_variance = variances[count / 2];
stats.max_variance = variances[count - 1];
```

4. **Libération Récursive**:
    - Nettoie la mémoire de l'ensemble de l'arbre.

```
void free_quadtree_recursive(qtree_node_t *node) {
    if (!node)
        return;
    for (int i = 0; i < 4; i++) {
        free_quadtree_recursive(node->children[i]);
    }
    free(node);
}
```

Choix Techniques Généraux:

1. **Gestion Récursive**:
    - Permet une décomposition naturelle de l'image en sous-régions homogènes.
2. **Efficacité des Calculs**:
    - L'utilisation de structures compactes (e.g., bitfields pour `e` et `u`) optimise la mémoire.
    - Le suivi de la progression garantit un retour utilisateur même sur de grands ensembles de données.
3. **Analyse Statistique**:

- Fournit des métriques clés sur la variabilité des données (variance médiane, variance maximale).

Module `pgm`

Rôle:

Le module `pgm` gère la lecture, l'écriture et la manipulation des images au format PGM (Portable Gray Map). Il offre des fonctions robustes pour valider les fichiers, lire leurs contenus, et les écrire correctement.

Détails du Fichier `pgm.h`

Rôle:

- Fournir une interface publique pour travailler avec des fichiers PGM.
- Gérer la validation des fichiers et des paramètres.

Contenu:

1. **Structure** `pgm_t` :
   - Représente une image PGM en mémoire :
     - Données des pixels (grayscale, 8 bits).
     - Taille (dimensions carrées).
     - Valeur maximale des pixels.

```
typedef struct {
    uint8_t *pixels;   /* Données des pixels */
    uint32_t size;     /* Taille de l'image (carrée) */
    uint8_t max_value; /* Valeur maximale des pixels */
} pgm_t;
```

2. **Enumération** `pgm_status_t` :
   - Indique les états possibles des opérations sur les fichiers PGM.

```
typedef enum {
    PGM_SUCCESS = 0,  /* Tout s'est bien passé */
```

```
    PGM_ERROR_FILE,   /* Problèmes avec le fichier */
    PGM_ERROR_FORMAT, /* Format invalide */
    PGM_ERROR_MEMORY, /* Mémoire insuffisante */
    PGM_ERROR_SIZE,   /* Taille non conforme */
    PGM_ERROR_PARAM   /* Paramètres incorrects */
} pgm_status_t;
```

3. **Fonctions Publiques**:
   - **Lecture**:
     - `pgm_status_t pgm_read(const char *path, pgm_t *pgm);`
   - **Écriture**:
     - `pgm_status_t pgm_write(const pgm_t *pgm, const char *path);`
   - **Gestion de la Mémoire**:
     - `void pgm_free(pgm_t *pgm);`

Détails du Fichier `pgm.c`

Rôle:

- Implémenter les fonctions de lecture, d'écriture et de validation des fichiers PGM.
- Offrir une gestion robuste des erreurs et de la mémoire.

Fonctionnalités Clés:

1. **Lecture des Fichiers**:
   - **Validation de l'En-Tête**:
     - Analyse du format ("P5" pour PGM binaire).
     - Validation des dimensions (carrées et puissance de 2).
     - Lecture de la valeur maximale des pixels (doit être ≤ 255).

```c
static pgm_status_t read_header(FILE *file, pgm_t *pgm) {
    char magic[3] = {0};
    unsigned int width, height, max_val;

    // Vérification du magic number
    if (fgets(magic, sizeof(magic), file) == NULL ||
        strncmp(magic, MAGIC_NUMBER, 2) != 0) {
        return PGM_ERROR_FORMAT;
    }
```

```
    // Lecture des dimensions et validation
    if (fscanf(file, "%u %u", &width, &height) != 2 || width !=
height || !is_power_of_two(width)) {
        return PGM_ERROR_SIZE;
    }
    pgm->size = width;

    // Lecture et validation de la valeur maximale
    if (fscanf(file, "%u", &max_val) != 1 || max_val > 255) {
        return PGM_ERROR_FORMAT;
    }
    pgm->max_value = (uint8_t)max_val;

    return PGM_SUCCESS;
}
```

- **Lecture des Pixels**:
  - Alloue un tampon pour les pixels.
  - Lit les données brutes après l'en-tête.

```
pgm_status_t pgm_read(const char *path, pgm_t *pgm) {
    FILE *file = fopen(path, "rb");
    if (!file) {
        return PGM_ERROR_FILE;
    }

    // Initialisation de la structure
    memset(pgm, 0, sizeof(pgm_t));

    // Lecture de l'en-tête
    pgm_status_t status = read_header(file, pgm);
    if (status != PGM_SUCCESS) {
        fclose(file);
        return status;
    }

    // Allocation du tampon pour les pixels
    size_t pixel_count = (size_t)pgm->size * pgm->size;
    pgm->pixels = malloc(pixel_count);
    if (!pgm->pixels) {
        fclose(file);
        return PGM_ERROR_MEMORY;
    }

    // Lecture des données de pixels
```

```
    if (fread(pgm->pixels, 1, pixel_count, file) != pixel_count) {
        pgm_free(pgm);
        fclose(file);
        return PGM_ERROR_FORMAT;
    }

    fclose(file);
    return PGM_SUCCESS;
}
```

2. **Écriture des Fichiers**:
   - Valide les paramètres et écrit l'en-tête suivie des pixels dans un fichier binaire.

```
pgm_status_t pgm_write(const pgm_t *pgm, const char *path) {
    FILE *file = fopen(path, "wb");
    if (!file) {
        return PGM_ERROR_FILE;
    }

    // Écriture de l'en-tête
    if (fprintf(file, "%s\n%u %u\n%u\n",
                MAGIC_NUMBER, pgm->size, pgm->size, pgm->max_value) <
0) {
        fclose(file);
        return PGM_ERROR_FILE;
    }

    // Écriture des données de pixels
    size_t pixel_count = (size_t)pgm->size * pgm->size;
    if (fwrite(pgm->pixels, 1, pixel_count, file) != pixel_count) {
        fclose(file);
        return PGM_ERROR_FILE;
    }

    fclose(file);
    return PGM_SUCCESS;
}
```

3. **Gestion de la Mémoire**:
   - Libère la mémoire associée à une image PGM.

```
void pgm_free(pgm_t *pgm) {
    if (pgm) {
        free(pgm->pixels);
```

```
        pgm->pixels = NULL;
        pgm->size = 0;
        pgm->max_value = 0;
    }
}
```

4. **Gestion des Commentaires**:
   - Ignore les commentaires et les espaces inutiles dans les fichiers PGM.

```
static int skip_ws_and_comments(FILE *file) {
    int c;

    while ((c = fgetc(file)) != EOF) {
        if (isspace(c)) {
            continue;
        }
        if (c == '#') {
            while ((c = fgetc(file)) != EOF && c != '\n')
                ;
            if (c == EOF)
                return -1;
        } else {
            ungetc(c, file);
            return 0;
        }
    }
    return -1;
}
```

Choix Techniques Généraux:

1. **Validation Rigoriste**:
   - Garantit que les fichiers lus respectent les contraintes (carré, puissance de 2, 8 bits).
2. **Séparation des Responsabilités**:
   - Chaque fonction gère une étape distincte : lecture d'en-tête, lecture des pixels, écriture, nettoyage.
3. **Gestion des Erreurs**:
   - Renvoie des codes d'état clairs pour simplifier le débogage et l'intégration avec d'autres modules.

Module `segmentation_grid`

Rôle:

Le module `segmentation_grid` est responsable de la génération d'une visualisation graphique d'un quadtree sous forme de grille, en divisant une image selon les régions définies par le quadtree. Le résultat est sauvegardé au format PGM.

Détails du Fichier `segmentation_grid.h`

Rôle:

- Fournir une interface publique pour créer des visualisations de quadtrees sous forme de grilles.
- Déclarer la fonction principale pour générer une grille.

Contenu:

1. **Fonction Publique**:
   - `qtree_status_t qtree_generate_grid(const qtree_t *tree, const char *output_file);`
     - Génère une image en niveau de gris montrant comment le quadtree divise les régions de l'image.
     - **Paramètres**:
       - `tree` : Le quadtree à visualiser.
       - `output_file` : Chemin du fichier où sauvegarder l'image.
     - **Retourne**:
       - `QTREE_SUCCESS` en cas de succès.
       - Un autre code d'erreur (`QTREE_ERROR_MEMORY` , `QTREE_ERROR_FORMAT` , etc.) en cas de problème.

Détails du Fichier `quadtree_grid.c`

Rôle:

- Implémenter la logique de génération d'une grille basée sur un quadtree.
- Offrir des fonctions internes pour dessiner les lignes horizontales et verticales de la grille.

Fonctionnalités Clés:

1. **Génération de Grilles**:
   - **Fonction Principale** `qtree_generate_grid`:
     - Crée une image PGM vide.
     - Trace les lignes de division de la grille en utilisant le quadtree.
     - Ajoute une bordure extérieure.
     - Sauvegarde l'image au format PGM.

```
qtree_status_t qtree_generate_grid(const qtree_t *tree, const char
*output_file) {
    if (!tree || !tree->root || !output_file) {
        return QTREE_ERROR_INVALID_PARAM;
    }

    // Créer une image PGM vide
    pgm_t grid_pgm;
    grid_pgm.size = tree->size;
    grid_pgm.max_value = 255;
    grid_pgm.pixels = calloc(tree->size * tree->size,
sizeof(uint8_t));

    if (!grid_pgm.pixels) {
        return QTREE_ERROR_MEMORY;
    }

    // Tracer les lignes de la grille
    draw_node_grid(grid_pgm.pixels, tree->size, tree->root, 0, 0,
tree->size);

    // Tracer la bordure extérieure
    draw_horizontal_line(grid_pgm.pixels, tree->size, 0, 0, tree-
>size);
    draw_horizontal_line(grid_pgm.pixels, tree->size, 0, tree->size -
1, tree->size);
    draw_vertical_line(grid_pgm.pixels, tree->size, 0, 0, tree-
>size);
    draw_vertical_line(grid_pgm.pixels, tree->size, tree->size - 1,
0, tree->size);

    // Sauvegarder l'image PGM
```

```
    pgm_status_t status = pgm_write(&grid_pgm, output_file);
    free(grid_pgm.pixels);

    return (status == PGM_SUCCESS) ? QTREE_SUCCESS :
QTREE_ERROR_FORMAT;
}
```

2. **Dessin des Lignes**:
   - **Lignes Horizontales (`draw_horizontal_line`)**:
     - Trace une ligne horizontale à partir d'une position donnée.

```
static void draw_horizontal_line(uint8_t *pixels, const size_t size,
                                 const size_t x, const size_t y,
                                 const size_t width) {
    for (size_t i = 0; i < width; i++) {
        for (size_t t = 0; t < GRID_LINE_THICKNESS; t++) {
            if (y + t < size) {
                pixels[(y + t) * size + x + i] = GRID_COLOR;
            }
        }
    }
}
```

   - **Lignes Verticales (`draw_vertical_line`)**:
     - Trace une ligne verticale à partir d'une position donnée.

```
static void draw_vertical_line(uint8_t *pixels, const size_t size,
                               const size_t x, const size_t y,
                               const size_t height) {
    for (size_t i = 0; i < height; i++) {
        for (size_t t = 0; t < GRID_LINE_THICKNESS; t++) {
            if (x + t < size) {
                pixels[(y + i) * size + x + t] = GRID_COLOR;
            }
        }
    }
}
```

3. **Dessin Récursif**:
   - `draw_node_grid`:
     - Parcourt récursivement le quadtree pour tracer les divisions correspondant aux enfants d'un nœud.

```c
static void draw_node_grid(uint8_t *pixels, const size_t size,
                           const qtree_node_t *node,
                           const size_t x, const size_t y,
                           const size_t node_size) {
    if (!node || node_size <= 1) {
        return;
    }

    if (!qtree_is_leaf(node)) {
        size_t half_size = node_size / 2;

        draw_horizontal_line(pixels, size, x, y + half_size,
node_size);
        draw_vertical_line(pixels, size, x + half_size, y,
node_size);

        draw_node_grid(pixels, size, node-
>children[QUADRANT_TOP_LEFT], x, y, half_size);
        draw_node_grid(pixels, size, node-
>children[QUADRANT_TOP_RIGHT], x + half_size, y, half_size);
        draw_node_grid(pixels, size, node-
>children[QUADRANT_BOTTOM_LEFT], x, y + half_size, half_size);
        draw_node_grid(pixels, size, node-
>children[QUADRANT_BOTTOM_RIGHT], x + half_size, y + half_size,
half_size);
    }
}
```

Choix Techniques Généraux:

1. **Approche Récursive**:
   - Permet de parcourir et de diviser efficacement l'image en fonction de la structure du quadtree.
2. **Optimisation de la Mémoire**:
   - Utilise un tableau dynamique pour représenter l'image, avec allocation et libération explicites.
3. **Gestion des Dimensions**:
   - Garantit que la taille des images est correcte et compatible avec les divisions récursives (puissance de 2).
4. **Visualisation Simple et Intuitive**:

- Ajout de couleurs fixes (mi-gris) pour représenter les lignes de la grille.
- Bordure extérieure pour améliorer la lisibilité.

Module `compression`

Rôle:

Le module `compression` implémente l'algorithme de compression sans perte pour les structures quadtree. Il prend en charge l'encodage des données à un niveau granulaire, en incluant des métadonnées, et offre une option de compression avec perte pour réduire davantage la taille des données.

Détails du Fichier `compression.h`

Rôle:

- Fournir une interface publique pour compresser les structures quadtree.
- Offrir des fonctions auxiliaires pour l'encodage et l'optimisation des données.

Contenu:

1. **Structure** `qtree_compress_state_t`:
   - Gère l'état de la compression, incluant :
     - Tampon temporaire pour les bits.
     - Position actuelle dans le tampon.
     - Nombre total de bits et d'octets traités.
     - Indicateur d'erreur.

```
typedef struct {
    uint8_t buffer;          /* Tampon pour les bits avant écriture */
    size_t bit_position;     /* Position actuelle dans le tampon */
    FILE *file;              /* Fichier de sortie */
    size_t bytes_written;    /* Octets écrits */
    size_t total_bits;       /* Bits totaux traités */
    int error;               /* Indicateur d'erreur */
    size_t total_nodes;      /* Nombre total de nœuds à traiter */
```

```
    size_t processed_nodes;  /* Nœuds déjà traités */
} qtree_compress_state_t;
```

2. **Fonctions Publiques**:
   - **Compression sans Perte**:
     - `qtree_compress_state_t compress_init(FILE *file);`
     - `void compress_write_bit(qtree_compress_state_t *state, uint8_t bit);`
     - `void compress_write_bits(qtree_compress_state_t *state, uint32_t value, size_t num_bits);`
     - `void compress_flush(qtree_compress_state_t *state);`
     - `float compress_get_rate(size_t total_bits, size_t original_size);`
     - `qtree_status_t compress(const qtree_t *tree, const char *output_filename, FILE *output_file);`
   - **Compression avec Perte**:
     - `qtree_status_t apply_lossy_compression(qtree_t *tree, float alpha);`

Détails du Fichier `quadtree_compress.c`

Rôle:

- Implémenter l'algorithme de compression sans perte et les méthodes associées.
- Fournir une option de compression avec perte pour réduire les variations faibles dans les données.

Fonctionnalités Clés:

1. **Compression Sans Perte**:
   - **Initialisation**:
     - La fonction `compress_init` prépare l'état de compression en initialisant les champs requis.

```
qtree_compress_state_t compress_init(FILE *file) {
    return (qtree_compress_state_t){
        .buffer = 0,
        .bit_position = 0,
        .file = file,
        .bytes_written = 0,
        .total_bits = 0,
        .error = 0,
```

```
        .total_nodes = 0,
        .processed_nodes = 0};
}
```

- **Écriture des Bits**:
  - Les bits sont accumulés dans un tampon avant d'être écrits dans le fichier pour optimiser les opérations d'E/S.

```
void compress_write_bit(qtree_compress_state_t *state, const uint8_t
bit) {
    state->buffer |= (bit & 1) << (7 - state->bit_position++);
    state->total_bits++;

    if (state->bit_position == 8) {
        if (fwrite(&state->buffer, 1, 1, state->file) != 1) {
            state->error = 1;
            return;
        }
        state->buffer = 0;
        state->bit_position = 0;
        state->bytes_written++;
    }
}
```

- **Compression des Niveaux**:
  - Chaque niveau de l'arbre est compressé récursivement en suivant l'ordre des quadrants.

```
static void compress_tree_level(qtree_compress_state_t *state, const
qtree_t *tree,
                                const qtree_node_t *node, const
uint32_t current_level,
                                const uint32_t target_level, const
bool is_interpolated) {
    if (node == NULL || state->error)
        return;

    const bool is_leaf = node->e == 0 && node->u == 1 &&
current_level == tree->n_levels;

    if (current_level == target_level) {
        write_node(state, node, is_leaf, is_interpolated);
        return;
    }
```

```
    if (node->u == 0) {
        for (int i = 0; i < 4; i++) {
            compress_tree_level(state, tree, node-
>children[quadrant_order[i]],
                                current_level + 1, target_level, i ==
3);
        }
    }
}
```

- **Calcul du Taux de Compression**:
  - Le taux de compression est calculé comme un pourcentage du ratio entre les bits compressés et les bits originaux.

```
float compress_get_rate(const size_t total_bits, const size_t
original_size) {
    return (float)total_bits / (float)original_size * 100.0f;
}
```

2. **Compression Avec Perte**:
   - **Filtrage Basé sur la Variance**:
     - Les nœuds avec une variance faible peuvent être combinés pour réduire la taille sans dégrader significativement les données.

```
static bool filter_node_recursive(qtree_node_t *node, float
threshold, float alpha) {
    if (!node || qtree_is_leaf(node))
        return true;

    update_node_variance(node);

    bool all_children_uniform = true;
    for (int i = 0; i < 4; i++) {
        if (node->children[i]) {
            if (!filter_node_recursive(node->children[i], threshold *
alpha, alpha)) {
                all_children_uniform = false;
            }
        }
    }

    if ((node->v <= threshold) && all_children_uniform) {
```

```
            node->u = 1;
            node->e = 0;
            for (int i = 0; i < 4; i++) {
                if (node->children[i]) {
                    free_quadtree_recursive(node->children[i]);
                    node->children[i] = NULL;
                }
            }
            return true;
        } else {
            node->u = is_uniform_block(node);
            return node->u;
        }
    }
```

Choix Techniques Généraux:

1. **Encodage Granulaire**:
   - Utilise un encodage au niveau du bit pour optimiser la taille du fichier compressé.
2. **Approche Récursive**:
   - Traverse l'arbre en profondeur pour encoder chaque niveau.
3. **Optimisation de la Mémoire**:
   - Accumule les bits dans un tampon avant de les écrire dans le fichier.
4. **Support des Variances**:
   - L'option avec perte permet d'ajuster dynamiquement les seuils en fonction de la variance locale.

Module `decompression`

Rôle:

Le module `decompression` est conçu pour restaurer les structures d'images compressées au format quadtree en utilisant une approche optimisée. Il permet de :

- Décompresser les fichiers quadtree compressés.
- Régénérer les images à partir des structures quadtree.
- Fournir un suivi des statistiques et une gestion des erreurs tout au long du processus.

Détails du Fichier `decompression.h`

Rôle:

- Fournir une interface claire pour les opérations de décompression.
- Convertir un quadtree en fichier PGM.

Contenu:

1. **Fonctions Principales** :
   - **Décompression** :
     - `qtree_status_t qtree_decompress(FILE *file, const char *input_filename, qtree_t *tree);`
     - Lit un fichier compressé et régénère l'arbre quadtree correspondant.
   - **Conversion en PGM** :
     - `qtree_status_t qtree_to_pgm(const qtree_t *tree, const char *output_filename, pgm_t *pgm);`
     - Construit une image à partir d'un quadtree.

Détails du Fichier `quadtree_decompress.c`

Rôle:

- Implémenter la logique de décompression des fichiers quadtree compressés.
- Gérer la conversion de quadtree en fichiers PGM avec un suivi statistique.

Fonctionnalités Clés:

1. **Décompression du Fichier**:
   - **Lecture de l'En-tête** :
     - Valide le fichier compressé (écriture et lecture des métadonnées).
     - Lit le niveau de profondeur de l'arbre.

     ```
     static void process_file_header(FILE *file, char *magic, uint8_t *levels);
     ```

   - **Décompression Récursive des Nœuds** :

- Reconstruit chaque nœud à partir des bits compressés, gère les états uniformes et les enfants.

```
static qtree_node_t *decompress_node(bit_reader_t *reader, uint32_t
level,
                                     uint32_t max_level, qtree_node_t
*parent,
                                     int child_index);
```

- **Gestion des Bits** :
  - Lit les bits compressés à l'aide d'un tampon avec suivi des erreurs.

```
static uint8_t read_bit(bit_reader_t *reader);
static uint8_t read_bits(bit_reader_t *reader, size_t num_bits);
```

2. **Conversion en PGM**:
   - **Extraction des Pixels** :
     - Parcourt le quadtree pour remplir une structure PGM en suivant l'ordre des quadrants.

```
static void extract_pixels(const qtree_node_t *node, uint8_t *pixels,
                           uint32_t row, uint32_t col,
                           uint32_t size, uint32_t total_size);
```

   - **Construction de l'Image** :
     - Initialise une structure PGM et alloue la mémoire pour les pixels.

```
qtree_status_t qtree_to_pgm(const qtree_t *tree, const char
*output_filename, pgm_t *pgm);
```

3. **Statistiques et Suivi**:
   - Suivi du progrès avec des mises à jour en temps réel et des journaux détaillés.
   - Calcul des ratios de compression, du temps CPU et des taux de traitement.

```
static void update_progress(decompress_stats_t *stats);
static decompress_stats_t init_stats(uint32_t max_levels, size_t
image_size);
```

Choix Techniques:

1. **Bit Reader Amélioré**:
   - Gestion des erreurs pour s'assurer que les fichiers malformés ou incomplets sont signalés immédiatement.

```
typedef struct {
    uint8_t buffer;
    size_t position;
    FILE *file;
    bool eof;
    decompress_stats_t *stats;
    bool has_error;
    const char *error_msg;
} bit_reader_t;
```

2. **Gestion Optimisée des Niveaux**:
   - Limite les allocations mémoires inutiles en comptant uniquement les nœuds non uniformes au précédent niveau.

```
static qtree_node_t **decompress_level(bit_reader_t *reader, uint32_t
level,
                                       uint32_t max_level, qtree_node_t
**prev_level,
                                       size_t prev_level_size);
```

3. **Conversion Flexible**:
   - La fonction `qtree_to_pgm` utilise une approche récursive pour remplir efficacement les pixels en exploitant les propriétés du quadtree.
4. **Statistiques Exhaustives**:
   - Suivi des bits lus, des nœuds traités et des niveaux parcourus.
   - Utilisation des journaux pour informer sur les ratios de compression et la progression.

Exemple d'Utilisation:

Décompresser un Fichier:

```
FILE *input_file = fopen("image.qtc", "rb");
if (!input_file) {
```

```
    perror("Erreur lors de l'ouverture du fichier");
    return;
}

qtree_t tree;
if (qtree_decompress(input_file, "image.qtc", &tree) != QTREE_SUCCESS) {
    fprintf(stderr, "Erreur lors de la décompression");
}
fclose(input_file);
```

Convertir un Quadtree en PGM:

```
pgm_t pgm;
if (qtree_to_pgm(&tree, "output.pgm", &pgm) != QTREE_SUCCESS) {
    fprintf(stderr, "Erreur lors de la conversion en PGM");
}

pgm_free(&pgm);
free_quadtree_recursive(tree.root);
```

Module `codec`

Rôle:

Le module `codec` constitue l'interface principale pour la compression et la décompression des images en utilisant les quadtrees. Il orchestre les opérations entre les différents modules (écriture/lecture de fichiers, manipulation des quadtrees, gestion des erreurs) pour fournir une solution complète de traitement d'images.

Détails du Fichier `codec.h`

Rôle:

- Déclarer les fonctions principales pour la compression et la décompression.
- Définir les codes d'état utilisés par le programme.

Contenu:

1. **Enumération** `codec_status_t` :
   - Définit les différents états que les fonctions peuvent renvoyer.

```
typedef enum {
    CODEC_SUCCESS = 0,          /* Tout s'est bien passé */
    CODEC_ERROR_INVALID_PARAM, /* Paramètres invalides */
    CODEC_ERROR_FILE_IO,        /* Problèmes de lecture/écriture de
fichiers */
    CODEC_ERROR_MEMORY,         /* Pas assez de mémoire */
    CODEC_ERROR_FORMAT          /* Format de fichier incorrect */
} codec_status_t;
```

2. **Fonctions Publiques**:
   - `codec_status_t codec_compress(const config_t *config);`
     - Lance l'opération de compression d'une image.
   - `codec_status_t codec_decompress(const config_t *config);`
     - Lance l'opération de décompression d'une image.
   - `const char *codec_status_string(codec_status_t status);`
     - Convertit un code d'état en un message lisible par l'utilisateur.

Détails du Fichier `codec.c`

Rôle:

- Implémenter les fonctions principales de compression et de décompression.
- Assurer la coordination entre les modules de lecture/écriture de fichiers, de manipulation des quadtrees, et de suivi des erreurs.

Fonctionnalités Clés:

1. **Conversion des Codes d'État**:
   - Deux fonctions internes permettent de convertir les codes d'état des modules `quadtree` et `pgm` vers des codes utilisables par `codec`.

```
static codec_status_t convert_qtree_status(qtree_status_t status);
static codec_status_t convert_pgm_status(pgm_status_t status);
```

2. **Compression**:

- La fonction `codec_compress` effectue les étapes suivantes :
  - Lit le fichier d'entrée au format PGM.
  - Construit un quadtree à partir des pixels de l'image.
  - Applique une compression sans perte ou avec perte (selon le paramètre `alpha`).
  - Sauvegarde les données compressées dans un fichier de sortie.

```
codec_status_t codec_compress(const config_t *config);
```

- Exemple d'utilisation interne :

```
pgm_status_t pgm_result = pgm_read(config->input_file, &pgm);
if (pgm_result != PGM_SUCCESS) {
    log_error("Failed to read PGM file");
    status = convert_pgm_status(pgm_result);
    goto cleanup;
}
```

3. **Décompression**:
   - La fonction `codec_decompress` effectue les étapes suivantes :
     - Lit les données compressées à partir d'un fichier.
     - Reconstruit le quadtree correspondant.
     - Convertit le quadtree en une image au format PGM.

```
codec_status_t codec_decompress(const config_t *config);
```

- Exemple d'utilisation interne :

```
op_status = qtree_decompress(input, config->input_file, &tree);
if (op_status != QTREE_SUCCESS) {
    log_error("Failed to read compressed data");
    status = convert_qtree_status(op_status);
    goto cleanup;
}
```

4. **Gestion des Erreurs**:
   - La fonction `codec_status_string` traduit les codes d'erreur en messages explicites.

```
const char *codec_status_string(codec_status_t status) {
    if (status < 0 || status >= sizeof(status_strings) /
```

```
sizeof(status_strings[0])) {
        return "Unknown error";
    }
    return status_strings[status];
}
```

5. **Intégration avec les Autres Modules**:
   - Utilise `pgm` pour la lecture et écriture d'images.
   - Utilise `quadtree` pour la manipulation des structures de données.
   - Utilise `logger` pour enregistrer les étapes clés de l'exécution.

Choix Techniques Généraux:

1. **Validation des Paramètres**:
   - Chaque fonction valide ses paramètres avant de procéder, avec des messages d'erreur explicites en cas de problème.
2. **Gestion de la Mémoire**:
   - Toutes les ressources (fichiers, quadtree, buffers) sont libérées dans un bloc `cleanup` pour garantir une exécution propre.

```
cleanup:
    if (output) fclose(output);
    if (tree_initialized) free_quadtree_recursive(tree.root);
    if (pgm_initialized) pgm_free(&pgm);
    return status;
```

3. **Logging Structuré**:
   - Utilisation du module `logger` pour consigner les étapes clés, les erreurs, et les statistiques d'exécution.
4. **Modularité**:
   - Le module `codec` agit comme un point d'entrée pour les opérations de compression et de décompression, tandis que les détails de bas niveau sont gérés par des modules spécialisés (e.g., `pgm`, `quadtree`).

Module `main`

Rôle:

Le fichier `main.c` constitue le point d'entrée principal du programme. Il gère l'initialisation des composants, l'analyse des arguments, et l'exécution des opérations principales comme la compression ou la décompression des images.

Détails du Fichier `main.c`

Rôle:

- Initialiser la configuration globale et le logger.
- Analyser les arguments de ligne de commande.
- Exécuter les opérations demandées (compression ou décompression).
- Fournir un suivi de l'exécution et des statistiques.

Fonctionnalités Clés:

1. **Initialisation**:
   - Configure le logger pour afficher des messages formatés avec des couleurs et des horodatages.

```
logger_configure((logger_config_t){
    .enabled = true,
    .use_colors = true,
    .show_timestamp = true});
```

2. **Analyse des Arguments**:
   - Utilise `cli_parse_arguments` pour valider et stocker les options dans une structure `config_t`.

```
if (!cli_parse_arguments(argc, argv, &config)) {
    log_error("Failed to parse command line arguments");
    return EXIT_FAILURE;
}
```

3. **Exécution des Opérations**:
   - Compression:

```
codec_status_t result = codec_compress(&config);
if (result != CODEC_SUCCESS) {
    log_error("Compression failed: %s", codec_status_string(result));
    status = EXIT_FAILURE;
}
```

- Décompression:

```
codec_status_t result = codec_decompress(&config);
if (result != CODEC_SUCCESS) {
    log_error("Decompression failed: %s",
codec_status_string(result));
    status = EXIT_FAILURE;
}
```

4. **Suivi du Temps d'Exécution**:
   - Calcule et affiche la durée totale d'exécution.

```
double elapsed = (double)(clock() - start_time) / CLOCKS_PER_SEC;
log_success("Operation completed in %.3f seconds", elapsed);
```

Choix Techniques:

1. **Structure Modulaire**:
   - Le `main` agit comme un coordonnateur en déléguant les opérations à des modules spécifiques.
2. **Robustesse**:
   - Gestion des erreurs à chaque étape pour garantir que le programme réagit gracieusement en cas de problème (e.g., fichiers manquants, arguments invalides).
3. **Flexibilité**:
   - Supporte à la fois la compression et la décompression avec des options supplémentaires comme la génération de grilles.

Makefile

Rôle:

Le Makefile gère la compilation, l'assemblage et le nettoyage du projet. Il définit des règles pour compiler les fichiers source, générer l'exécutable, et nettoyer les fichiers intermédiaires.

Détails du Makefile

Structure Principale:

1. **Configuration du Projet**:
   - Nom du projet, version, et chemins de préfixe.

```
NAME        := codec
VERSION     := 1.0.0
PREFIX      ?= /usr/local
DESTDIR     ?=
```

2. **Définitions des Dossiers**:
   - Organisation des fichiers source, en-têtes et objets.

```
SRC_DIR     := src
INC_DIR     := include
BUILD_DIR   := build
```

3. **Compilation et Assemblage**:
   - Définit les fichiers source (`SRCS`), objets (`OBJS`), et dépendances (`DEPS`).

```
SRCS        := $(shell find $(SRC_DIR) -name '*.c')
OBJS        := $(SRCS:$(SRC_DIR)/%.c=$(BUILD_DIR)/%.o)
```

4. **Outils et Commandes**:
   - Utilise GCC pour la compilation et inclut des drapeaux de sécurité et d'optimisation.

```
CC          := gcc
CFLAGS      := -Wall -Wextra -O2 -std=c11 -pedantic-errors -I$(INC_DIR)
LDFLAGS     := -lm
```

Règles Principales:

1. **Compilation Complète**:

- Compile tous les fichiers source et génère l'exécutable final.

```
all: directories $(NAME)
$(NAME): $(OBJS)
    $(CC) $(OBJS) $(LDFLAGS) -o $@
```

2. **Nettoyage**:
   - Supprime les fichiers intermédiaires et l'exécutable.

```
clean:
    rm -rf $(BUILD_DIR) $(NAME)
```

3. **Règle par Défaut**:
   - Crée les sous-dossiers pour les fichiers objets.

```
directories:
    mkdir -p $(BUILD_DIR)
```

Choix Techniques:

1. **Optimisation**:
   - Les drapeaux d'optimisation (`-O2` , `-fstack-protector-strong`) assurent une bonne performance tout en maintenant la sécurité.
2. **Modularité**:
   - La structure des dossiers facilite l'ajout de nouveaux modules ou composants.
3. **Compatibilité**:
   - S'appuie sur des standards modernes (C11) et POSIX pour une portabilité accrue.

Enjoy yannis, j'espère c'est assez détaillé :D