Projet Réseaux 2

Jacques KOZIK

16 mai 2025

Table des matières

Introduction

Ce rapport présente mon projet développé dans le cadre du cours de Réseaux 2. L'objectif principal de ce projet est de créer une application permettant de visualiser et de manipuler des données binaires avec une interface graphique intuitive.

Le projet se compose de deux parties principales :

— Un module de traitement des données binaires, implémenté en C

— Une interface graphique développée avec la bibliothèque Raylib, offrant une visualisation claire des données

Cette approche permet de combiner les performances du langage C pour la manipulation des données avec une représentation graphique moderne et cross-platform. Raylib a été choisi pour sa simplicité d'utilisation et sa capacité à fonctionner sur différents systèmes d'exploitation (Windows, Linux, MacOS), ce qui m'a permis de me concentrer sur les algorithmes sans me perdre dans la complexité de l'interface graphique.

Dans ce rapport, je détaillerai l'architecture du projet, les choix techniques effectués, les fonctionnalités implémentées, ainsi que la façon dont les données sont transmises, reçues et interprétées par l'application.

Ce rapport s'organise en deux grandes parties. Dans un premier temps, je présenterai en détail les fonctions du module `binaire.c/binaire.h`, qui constitue le cœur du traitement des données binaires et de leur transformation. J'expliquerai les algorithmes utilisés pour la manipulation des données, notamment le code de Hamming pour la détection et correction d'erreurs. Dans un second temps, j'aborderai le fonctionnement de l'interface graphique développée avec Raylib, qui permet de visualiser ces données de manière intuitive et interactive. Cette partie mettra en évidence les mécanismes d'encodage et décodage des messages en images, ainsi que les techniques employées pour optimiser l'affichage et l'interaction utilisateur.

1 Le module binaire

1.1 Vue d'ensemble du module

Le module `binaire.c/binaire.h` constitue le cœur algorithmique du projet. Il implémente l'ensemble des fonctions nécessaires à la manipulation des données binaires, aux conversions entre différentes représentations (ASCII, binaire, chaînes de caractères), et à l'application du code de Hamming pour la détection et correction d'erreurs.

Ce module est conçu avec une approche modulaire, où chaque fonction remplit un rôle précis et bien défini. Cette architecture permet une réutilisation efficace des composants dans différentes parties du programme.

1.2 Conversions et représentations

1.2.1 Fonctions de conversion ASCII-binaire

Le module implémente plusieurs fonctions de base pour convertir entre les différentes représentations :

— `Char_vers_ASCII` et `ASCII_vers_Char` : Fonctions simples de conversion entre un caractère et son code ASCII correspondant

```
int Char_vers_ASCII(char l){
    return (int)l;
}

```

Listing 1 – Fonction Char_vers_ASCII

```
char ASCII_vers_Char(int code){
    return (char)code;
}

```

Listing 2 – Fonction ASCII_vers_Char

— `ASCII_vers_Binaire` : Convertit un code ASCII (entier) en sa représentation binaire sur 8 bits

```
int* ASCII_vers_Binaire(int code){
    int* binaire = malloc(8 * sizeof(int));
    for (int i = 7; i >= 0; i--) {
        binaire[i] = code % 2;
        code = code / 2;
    }
    return binaire;
}

```

Listing 3 – Fonction ASCII_vers_Binaire

— `Binaire_vers_ASCII` : Effectue l'opération inverse, en reconstruisant un code ASCII à partir d'une séquence de 8 bits

```
int Binaire_vers_ASCII(int* binaire){
    int ascii = 0;
    for (int i = 0; i < 8; i++) {
        if(binaire[i]==1){
            ascii += puissance(2,7-i);
```

```
        }
    }
    return ascii;
}

```

Listing 4 – Fonction Binaire_vers_ASCII

1.2.2 Manipulation de chaînes et tableaux binaires

Pour traiter des messages complets, le module propose des fonctions de plus haut niveau :

— `String_vers_Binaire` : Convertit une chaîne de caractères complète en sa représentation binaire (tableau de tableaux de bits)

```
int** String_vers_Binaire(char* chaine, int taille){
    int** binaire = Init_Tableau_Binaire_Vide(taille);
    for(int i=0; i<taille;i++){
            int code = Char_vers_ASCII(chaine[i]);
            binaire[i] = ASCII_vers_Binaire(code);
    }
    return binaire;
}

```

Listing 5 – Fonction String_vers_Binaire

— `Binaire_vers_String` : Reconstitue une chaîne de caractères à partir de sa représentation binaire

```
char* Binaire_vers_String(int** binaire, int taille){
    char* mot = malloc((taille + 1) * sizeof(char));

    int ascii;
    for(int i = 0; i < taille; i++){
        ascii = Binaire_vers_ASCII(binaire[i]);
        mot[i] = ASCII_vers_Char(ascii);
    }

    mot[taille] = '\0';

    return mot;
}

```

Listing 6 – Fonction Binaire_vers_String

— `Affichage_Binaire` et `Affichage_String` : Utilitaires pour l'affichage formaté dans le terminal

```c
void Affichage_Binaire(int** binaire, int taille){
    for(int i=0;i<taille;i++){
        for(int j=0;j<8;j++){
            printf("%d",binaire[i][j]);
        }
        printf("\n");
    }
}

```

Listing 7 – Fonction Affichage_Binaire

```c
void Affichage_String(char* phrase, int taille){
    for(int i = 0; i<taille;i++){
        printf("%c", phrase[i]);
    }
}

```

Listing 8 – Fonction Affichage_String

1.3 Implémentation du code de Hamming

Le cœur technique du module réside dans son implémentation du code de Hamming, un algorithme de détection et correction d'erreurs particulièrement adapté à notre contexte.

1.3.1 Principe du code de Hamming

Le code de Hamming utilisé (Hamming(8,4)) prend 4 bits de données et génère 4 bits de parité supplémentaires, permettant de détecter et corriger des erreurs simples (un seul bit erroné). Dans notre implémentation :

- Les caractères sont encodés sur 8 bit en langage C, donc ils sont divisé en deux sous-parties
- Les bits de données sont placés aux positions 3, 5, 6 et 7 (indices 2, 4, 5, 6 en base zéro)
- Les bits de parité sont calculés et placés aux positions 1, 2, 4 et 8 (indices 0, 1, 3, 7)
- Le bit de parité global (position 8) permet de détecter si une erreur double s'est produite

1.3.2 Fonctions d'encodage et décodage

Les fonctions principales liées au code de Hamming sont :

— `Binaire_vers_Code_Hamming` : Encode une séquence binaire (représentant un message) en utilisant le code de Hamming

```
int** Binaire_vers_Code_Hamming(int** binaire, int taille) {
    int** hamming_code = Init_Tableau_Binaire_Vide(2 * taille);
    int k;

    for (int i = 0; i < taille; i++) {
        k = 2 * i;

        int d1 = binaire[i][0];
        int d2 = binaire[i][1];
        int d3 = binaire[i][2];
        int d4 = binaire[i][3];
        int d5 = binaire[i][4];
        int d6 = binaire[i][5];
        int d7 = binaire[i][6];
        int d8 = binaire[i][7];

        hamming_code[k][2] = d1;
        hamming_code[k][4] = d2;
        hamming_code[k][5] = d3;
        hamming_code[k][6] = d4;

        hamming_code[k][0] = (d1 + d2 + d4) % 2; // P1
        hamming_code[k][1] = (d1 + d3 + d4) % 2; // P2
        hamming_code[k][3] = (d2 + d3 + d4) % 2; // P4
        hamming_code[k][7] = (hamming_code[k][0] + hamming_code[k][1] + hamming_code[k][3]
                              + d1 + d2 + d3 + d4) % 2;

        hamming_code[k + 1][2] = d5;
        hamming_code[k + 1][4] = d6;
        hamming_code[k + 1][5] = d7;
        hamming_code[k + 1][6] = d8;

        hamming_code[k + 1][0] = (d5 + d6 + d8) % 2;
        hamming_code[k + 1][1] = (d5 + d7 + d8) % 2;
        hamming_code[k + 1][3] = (d6 + d7 + d8) % 2;
        hamming_code[k + 1][7] = (hamming_code[k + 1][0] + hamming_code[k + 1][1]
                                  + hamming_code[k + 1][3] + d5 + d6 + d7 + d8) % 2;
    }

    return hamming_code;
}
```

```
42
```

Listing 9 – Fonction Binaire_vers_Code_Hamming

— `Binaire_Simple_vers_Code_Hamming` : Version simplifiée traitant un seul octet

```
int** Binaire_Simple_vers_Code_Hamming(int* binaire) {
    const int taille = 2;
    int** hamming_code = Init_Tableau_Binaire_Vide(2);

    int d1 = binaire[0];
    int d2 = binaire[1];
    int d3 = binaire[2];
    int d4 = binaire[3];
    int d5 = binaire[4];
    int d6 = binaire[5];
    int d7 = binaire[6];
    int d8 = binaire[7];

    hamming_code[0][2] = d1;
    hamming_code[0][4] = d2;
    hamming_code[0][5] = d3;
    hamming_code[0][6] = d4;

    hamming_code[0][0] = (d1 + d2 + d4) % 2;
    hamming_code[0][1] = (d1 + d3 + d4) % 2;
    hamming_code[0][3] = (d2 + d3 + d4) % 2;
    hamming_code[0][7] = (hamming_code[0][0] +
   hamming_code[0][1] + hamming_code[0][3]
                          + d1 + d2 + d3 + d4) % 2;

    hamming_code[1][2] = d5;
    hamming_code[1][4] = d6;
    hamming_code[1][5] = d7;
    hamming_code[1][6] = d8;

    hamming_code[1][0] = (d5 + d6 + d8) % 2;
    hamming_code[1][1] = (d5 + d7 + d8) % 2;
    hamming_code[1][3] = (d6 + d7 + d8) % 2;
    hamming_code[1][7] = (hamming_code[1][0] +
   hamming_code[1][1] + hamming_code[1][3]
                          + d5 + d6 + d7 + d8) % 2;

    return hamming_code;
}
```

Listing 10 – Fonction Binaire_Simple_vers_Code_Hamming

— `Code_Hamming_vers_Binaire` : Décode les données encodées avec Hamming et reconstruit le message original

```c
int** Code_Hamming_vers_Binaire(int** hamming_code, int taille){
    printf("Code_Hamming_vers_Binaire: debut avec taille=%d\n", taille);

    int** binaire = Init_Tableau_Binaire_Vide(taille/2);

    int k;
    for(int i=0; i<taille/2; i++){
        k = 2*i;

        if (k >= taille || k+1 >= taille) {
            printf("Erreur: indice hors limites k=%d (taille=%d)\n", k, taille);
            Liberer_Tableau_Binaire(binaire, taille/2);
            return NULL;
        }

        if (i == 0) {
            printf("Premier hamming_code[%d]: ", k);
            for (int j = 0; j < 8; j++) {
                printf("%d ", hamming_code[k][j]);
            }
            printf("\n");
        }

        binaire[i][0] = hamming_code[k][2];
        binaire[i][1] = hamming_code[k][4];
        binaire[i][2] = hamming_code[k][5];
        binaire[i][3] = hamming_code[k][6];
        binaire[i][4] = hamming_code[k+1][2];
        binaire[i][5] = hamming_code[k+1][4];
        binaire[i][6] = hamming_code[k+1][5];
        binaire[i][7] = hamming_code[k+1][6];
    }

    printf("Code_Hamming_vers_Binaire: decodage reussi\n");
    return binaire;
}

```

Listing 11 – Fonction Code_Hamming_vers_Binaire

— `Code_Hamming_vers_Binaire_Simple` : Version simplifiée pour un seul octet

```
int* Code_Hamming_vers_Binaire_Simple(int** hamming_code
    ){
     int* binaire = malloc(8*sizeof(int));
     binaire[0] = hamming_code[0][2];
     binaire[1] = hamming_code[0][4];
     binaire[2] = hamming_code[0][5];
     binaire[3] = hamming_code[0][6];
     binaire[4] = hamming_code[1][2];
     binaire[5] = hamming_code[1][4];
     binaire[6] = hamming_code[1][5];
     binaire[7] = hamming_code[1][6];
     return binaire;
}

```

Listing 12 – Fonction Code_Hamming_vers_Binaire_Simple

L'implémentation suit un modèle mathématique précis où chaque bit de parité est calculé en fonction des positions spécifiques des bits de données qu'il doit protéger.

1.4 Gestion de la mémoire

Le module implémente une gestion rigoureuse de la mémoire avec :

— `Init_Tableau_Binaire_Vide` : Alloue dynamiquement la mémoire pour stocker les représentations binaires

```
int** Init_Tableau_Binaire_Vide(int taille){
    int** binaire = malloc(taille*sizeof(int*));
        for(int i = 0;i<taille;i++){
            binaire[i] = malloc(8*sizeof(int));
        }
    return binaire;
}

```

Listing 13 – Fonction Init_Tableau_Binaire_Vide

— `Liberer_Tableau_Binaire` : Libère proprement la mémoire allouée pour éviter les fuites

```
    void Liberer_Tableau_Binaire(int** tableau, int n) {
        if (tableau == NULL) return;

        for (int i = 0; i < n; i++) {
            if (tableau[i] != NULL) {
                free(tableau[i]);
            }
        }

```

```
10          free(tableau);
11      }
12
```

Listing 14 – Fonction Liberer_Tableau_Binaire

Cette approche permet de traiter des messages de taille variable tout en optimisant l'utilisation de la mémoire.

2 L'interface graphique avec Raylib

2.1 Introduction à Raylib

Raylib est une bibliothèque de développement graphique simple et légère, écrite en C. J'ai choisi cette bibliothèque pour plusieurs raisons :

— Sa simplicité d'utilisation et sa faible courbe d'apprentissage
— Son approche minimaliste et sa compatibilité multiplateforme
— Son intégration aisée avec le code C existant
— Ses fonctionnalités graphiques adaptées à la visualisation de données

L'objectif principal était de créer une interface permettant de visualiser les données binaires sous forme d'images et de pouvoir reconstruire le message original à partir de ces images.

2.2 Structure de l'application

L'application est organisée autour de deux modes principaux :

— Conversion d'un message texte en image binaire
— Conversion d'une image binaire en message texte

Voici l'initialisation et la structure principale du programme :

```
1  int main(void){
2    const int BUFF_SIZE = 256;
3    char buffer[BUFF_SIZE];
4    char input[BUFF_SIZE];
5
6    printf("---\n");
7    printf("Souhaitez vous (0) Convertir un message en image ou
       (1) convertir une image en message?\n");
8    printf("---\n");
9    fgets(input, BUFF_SIZE, stdin);
10
11   if(input[0]=='0'){
12     // Mode conversion message -> image
13   }
14   else{
```

```
15      // Mode conversion image -> message
16    }
17      return 0;
18  }
```

Listing 15 – Structure principale du programme

2.3 Conversion de texte en image

Cette partie de l'application permet à l'utilisateur de saisir un texte qui sera converti en représentation binaire, puis visualisé sous forme d'image.

```
1  printf("---\n");
2  printf("Quelle texte souhaiter vous transformer?\n");
3  printf("---\n");
4  fgets(buffer, BUFF_SIZE, stdin);
5
6  bool flag = true;
7  int taille_m = 0;
8  while(taille_m < BUFF_SIZE-1 && flag) {
9      if(buffer[taille_m] == '\n') {
10         flag = false;
11         buffer[taille_m] = '\0';
12         taille_m--;
13     }
14     taille_m++;
15 }
16
17 int** representation_binaire = String_vers_Binaire(buffer,
       taille_m);
18 int** representation_binaire_avec_code_hamming =
       Binaire_vers_Code_Hamming(representation_binaire,taille_m)
       ;
```

Listing 16 – Lecture et préparation du message

La taille du message est également encodée pour permettre le décodage ultérieur :

```
1  int* representation_binaire_taille = ASCII_vers_Binaire(
       taille_m);
2  int** representation_binaire_taille_avec_code_hamming =
       Binaire_Simple_vers_Code_Hamming(
       representation_binaire_taille);
3
4  const int BIT_MESSAGE = taille_m*2*16;
5  const int BIT_DEPART_ARRIVEE = 3;
6  const int BIT_TAILLE = 16;
7
```

```
const int NOMBRE_DE_BIT = BIT_MESSAGE + BIT_DEPART_ARRIVEE +
    BIT_TAILLE;
int ScreenSize = 0;
while((ScreenSize*ScreenSize)<NOMBRE_DE_BIT){
    ScreenSize++;
}
```

Listing 17 – Encodage de la taille du message

2.3.1 Visualisation graphique avec Raylib

L'initialisation de la fenêtre Raylib et la configuration de l'affichage :

```
const int SCREENWIDTH = ScreenSize*10;
const int SCREENHEIGHT = (ScreenSize)*10;

InitWindow(SCREENWIDTH, SCREENHEIGHT, "");
SetTargetFPS(60);
```

Listing 18 – Initialisation de Raylib

Le rendu de l'image binaire suit une approche structurée avec des marqueurs spécifiques :

```
BeginDrawing();
ClearBackground(RAYWHITE);

DrawRectangle(0,0,10,10,RED);

int X=0, Y=1, cpt=1;
while(cpt<NOMBRE_DE_BIT){
    for(int i=0;i<2;i++){
        for(int j=0;j<8;j++){
            if(Y==ScreenSize){
                Y=0;
                X++;
            }
            if(
    representation_binaire_taille_avec_code_hamming[i][j]==0){
                DrawRectangle(Y*10, X*10, 10, 10, WHITE);
                DrawRectangleLines(Y*10, X*10, 10, 10, WHITE)
    ;
            }
            else{
                DrawRectangle(Y*10, X*10, 10, 10, BLACK);
                DrawRectangleLines(Y*10, X*10, 10, 10, BLACK)
    ;
            }
            Y++;
            cpt++;
```

```
24          }
25      }
26
27      if(Y==ScreenSize){
28              Y=0;
29              X++;
30          }
31          DrawRectangle(Y*10, X*10, 10, 10, GREEN);
32          DrawRectangleLines(Y*10, X*10, 10, 10, GREEN);
33          Y++;
34          cpt++;
35      }
36
37      for(int i_tab=0;i_tab<taille_m*2;i_tab++){
38          for(int j_tab=0; j_tab<8;j_tab++){
39              if(Y==ScreenSize){
40                  Y=0;
41                  X++;
42              }
43
44              if(representation_binaire_avec_code_hamming[i_tab
    ][j_tab]==0){
45                  DrawRectangle(Y*10, X*10, 10, 10, WHITE);
46                  DrawRectangleLines(Y*10, X*10, 10, 10, WHITE)
    ;
47              }
48              else{
49                  DrawRectangle(Y*10, X*10, 10, 10, BLACK);
50                  DrawRectangleLines(Y*10, X*10, 10, 10, BLACK)
    ;
51              }
52
53              Y++;
54              cpt++;
55          }
56      }
57
58      if(Y==ScreenSize){
59          Y=0;
60          X++;
61      }
62      DrawRectangle(Y*10, X*10, 10, 10, BLUE);
63      DrawRectangleLines(Y*10, X*10, 10, 10, BLUE);
64  }
65
66  DrawRectangleLines(0, 0, SCREENWIDTH, SCREENHEIGHT, BLACK);
```

Listing 19 – Rendu des données binaires

2.3.2 Capture d'écran et sauvegarde

L'application permet de sauvegarder l'image générée sous forme de fichier PNG :

```
if (IsKeyPressed(KEY_S)) {
    for(int i = 0; i < 5; i++) {
        BeginDrawing();
        EndDrawing();
    }
    TakeScreenshot("code.png");
    printf("Screenshot taken");
}
```

Listing 20 – Capture d'écran et sauvegarde

2.4 Conversion d'image en texte

La seconde fonctionnalité majeure permet de décoder un message à partir d'une image contenant des données binaires.

```
printf("---\n");
printf("Quel est le nom de votre fichier?\n");
printf("---\n");
fgets(buffer, BUFF_SIZE, stdin);

bool flag = true;
int length = 0;
while(length < BUFF_SIZE-1 && flag) {
    if(buffer[length] == '\n') {
        flag = false;
        buffer[length] = '\0';
    }
    length++;
}

Image image = LoadImage(buffer);
int ScreenSize = image.width / 10;

Color startColor = GetImageColor(image, 5, 5);
if (!(startColor.r > 200 && startColor.g < 100 && startColor.
   b < 100)) {
    printf("Start marker not found in the image\n");
    UnloadImage(image);
    return 1;
}
```

Listing 21 – Chargement de l'image et vérification du marqueur de début

2.4.1 Lecture de la taille du message

Le décodage commence par extraire la taille encodée du message :

```
int X = 0, Y = 1;

int** taille_code_hamming = Init_Tableau_Binaire_Vide(2);
for(int i = 0; i < 2; i++) {
    for(int j = 0; j < 8; j++) {
        if(Y >= ScreenSize) {
            Y = 0;
            X++;
        }
        Color pixelColor = GetImageColor(image, Y * 10 + 5, X
    * 10 + 5);

        int brightness = (pixelColor.r + pixelColor.g +
   pixelColor.b) / 3;

        taille_code_hamming[i][j] = (brightness > 127) ? 0 :
   1;

        Y++;
    }
}

int* taille_bits = Code_Hamming_vers_Binaire_Simple(
   taille_code_hamming);
int taille_m = Binaire_vers_ASCII(taille_bits);
```

Listing 22 – Décodage de la taille du message

2.4.2 Vérification du marqueur séparateur

L'application recherche le marqueur vert qui sépare la taille du contenu du message :

```
if(Y >= ScreenSize) {
    Y = 0;
    X++;
}

Color sepColor = GetImageColor(image, Y * 10 + 5, X * 10 + 5)
   ;
if (!(sepColor.g > 180 && sepColor.r < 120 && sepColor.b <
   120)) {
    bool found = false;
    for (int testY = Y-1; testY <= Y+1 && !found; testY++) {
        for (int testX = X-1; testX <= X+1 && !found; testX
   ++) {
```

```
            if (testY >= 0 && testX >= 0 && testY <
    ScreenSize) {
                Color testColor = GetImageColor(image, testY
    * 10, testX * 10);
                if (testColor.g > 180 && testColor.r < 120 &&
     testColor.b < 120) {
                    X = testX;
                    Y = testY;
                    found = true;
                }
            }
        }
    }

    if (!found) {
        UnloadImage(image);
        return 1;
    }
}
Y++;
```

Listing 23 – Recherche du marqueur séparateur

2.4.3 Lecture et décodage du message

Enfin, l'application lit les données binaires et les décode en message texte :

```
int** message_bits = (int**)malloc(taille_m * 4 * sizeof(int
    *));

for(int i = 0; i < taille_m * 4; i++) {
    message_bits[i] = (int*)malloc(8 * sizeof(int));

    for(int j = 0; j < 8; j++) {
        if(Y >= ScreenSize) {
            Y = 0;
            X++;
        }

        Color pixelColor = GetImageColor(image, Y * 10 + 5, X
     * 10 + 5);
        message_bits[i][j] = (pixelColor.r > 180) ? 0 : 1;

        Y++;
    }
}

int** decoded_message = Code_Hamming_vers_Binaire(
    message_bits, taille_m*4);
```

```
20 char* result = Binaire_vers_String(decoded_message, taille_m)
       ;
21
22 printf("Decoded message: %s\n", result);
```

Listing 24 – Lecture et décodage du message

2.5 Avantages de l'approche adoptée

L'utilisation de Raylib et l'approche graphique pour la représentation des données binaires offrent plusieurs avantages :

- **Visualisation intuitive** : La représentation visuelle permet de mieux comprendre les données binaires
- **Format standard** : L'utilisation d'images PNG offre un format standardisé pour le stockage et la transmission
- **Correction d'erreurs intégrée** : Le code de Hamming permet de détecter et corriger des erreurs éventuelles dans l'image
- **Métadonnées incluses** : L'encodage de la taille du message permet un décodage précis sans information externe

2.6 Conclusion sur l'interface graphique

L'implémentation avec Raylib offre une solution élégante au problème de visualisation des données binaires. Les marqueurs colorés (rouge pour le début, vert pour le séparateur et bleu pour la fin) facilitent la lecture automatisée de l'image, tandis que l'encodage de Hamming assure une résistance aux erreurs lors de la transmission ou du stockage.

Cette approche pourrait être étendue à l'avenir pour inclure d'autres types de données ou pour améliorer la densité d'information en utilisant des couleurs plutôt que simplement du noir et blanc pour représenter plusieurs bits par pixel.

3 Compilation et Exécution du Projet

Pour compiler et exécuter ce projet sur votre machine, suivez ces étapes simples :

3.1 Prérequis

Avant de commencer, assurez-vous que votre système dispose des éléments suivants :

— Un compilateur C (gcc recommandé)
— La bibliothèque Raylib installée
— Make pour la compilation automatisée

3.2 Installation de Raylib

Pour installer Raylib, suivez les instructions spécifiques à votre système d'exploitation dans le fichier `README.md` du projet.

3.3 Compilation

Pour compiler le projet :

```
# Se placer dans le repertoire GUI
cd /chemin/vers/GUI

# Compiler le projet avec make
make
```

Cette commande génère l'exécutable dans le dossier `bin/Debug/`.

3.4 Exécution

Pour lancer l'application :

```
# Depuis le repertoire GUI
./bin/Debug/GUI
```

Une fois l'application lancée, suivez les instructions à l'écran pour convertir un message en image ou inversement.

3.5 Utilisation

1. **Conversion de texte en image** :
   — Sélectionnez l'option "0" lorsque l'invite vous demande quelle opération effectuer
   — Entrez le texte que vous souhaitez encoder
   — Une fois l'image générée, appuyez sur la touche "S" pour sauvegarder l'image sous le nom "code.png"
2. **Conversion d'image en texte** :
   — Sélectionnez l'option "1" lorsque l'invite vous demande quelle opération effectuer
   — Entrez le chemin vers le fichier image à décoder
   — Le message décodé s'affichera dans la console

Pour quitter l'application, fermez la fenêtre ou appuyez sur la touche Échap.