

TRƯỜNG ĐẠI HỌC ĐÀ LẠT
KHOA TOÁN - TIN HỌC



TIỂU LUẬN MÔN HỌC
Lập trình Hướng đối tượng (TN2301D)
ĐỀ TÀI:
**Viết game Tetris chạy trên
giao diện dòng lệnh (Console User Interface)**

SINH VIÊN THỰC HIỆN:
TRẦN DUY THANH
MSSV 2015830
LỚP TNK44

Đà Lạt, 29/09/2021

TÓM TẮT

Năm 1984, Alexey Pajitnov - một kĩ sư phần mềm người Liên Xô - đã viết ra Tetris. Tetris trở nên phổ biến trong tuổi thơ của nhiều lập trình viên kể từ đó. Ngày nay, nhiều lập trình viên vẫn thường viết lại Tetris như là một cách luyện tập sử dụng ngôn ngữ lập trình mới.

Nhìn chung, Tetris luật chơi dễ dàng, logic không quá phức tạp, khá thuận tiện để minh họa lập trình hướng đối tượng. Trong tiểu luận này, tôi sẽ lần lượt đi qua các mục sau:

1. Giới thiệu luật chơi cơ bản của Tetris sử dụng trong tiểu luận này.
2. Ý tưởng lập trình Tetris trên [giao diện dòng lệnh](#) và một số kĩ thuật trong ngôn ngữ C# liên quan đến ý tưởng này.
3. Tổng quan cấu trúc chương trình.

Mục lục

Mục lục	ii
1 Tổng quan về trò chơi Tetris	1
1.1 Tên gọi	1
1.2 Ý tưởng	1
2 Ý tưởng giải thuật, các bài toán cốt lõi	3
2.1 Về giao diện dòng lệnh (Console User Interface - CUI)	3
2.2 Các ký tự đặc biệt trong giao diện dòng lệnh	4
2.3 Xây dựng các khối hình và các biến thẻ xoay	6
2.3.1 Các kiểu xoay	9
2.4 Tạo chuyển động trong môi trường dòng lệnh	10
2.4.1 Chuyển động rơi	12
2.4.2 Dịch trái - phải	13
2.4.3 Xoay	14
2.5 Bắt và xử lý sự kiện gõ phím	15
2.6 Xử lí song song theo luồng (thread)	15
2.7 Sinh khối hình ngẫu nhiên kế tiếp	19
2.7.1 Kiểm tra chồng lấn	20
2.7.2 Kiểm tra mép trái, mép phải, mép dưới	23
2.7.3 Kiểm tra khả năng xoay khi gần mép	24
2.7.4 Kiểm tra hàng đầy, tính điểm và xóa hàng đầy	25
3 Tổng quan cấu trúc chương trình	28
3.1 Khai báo lớp đối tượng Block	28
3.2 Khai báo lớp đối tượng Background	31
3.3 Khai báo lớp đối tượng Screen	33
Tham khảo	37

Phần 1

Tổng quan về trò chơi Tetris

1.1 Tên gọi

Ông lấy tên của trò chơi từ tiền tố "tetra-" của tiếng Hy Lạp, có nghĩa là "bốn" (mỗi **khối hình** trong trò chơi đều gồm 4 **khối vuông cơ bản** xếp lại với nhau, nên ông gọi là Tetromino) và quần vợt (tennis - trò thể thao Pajitnov thích nhất) ghép lại thành Tetris.

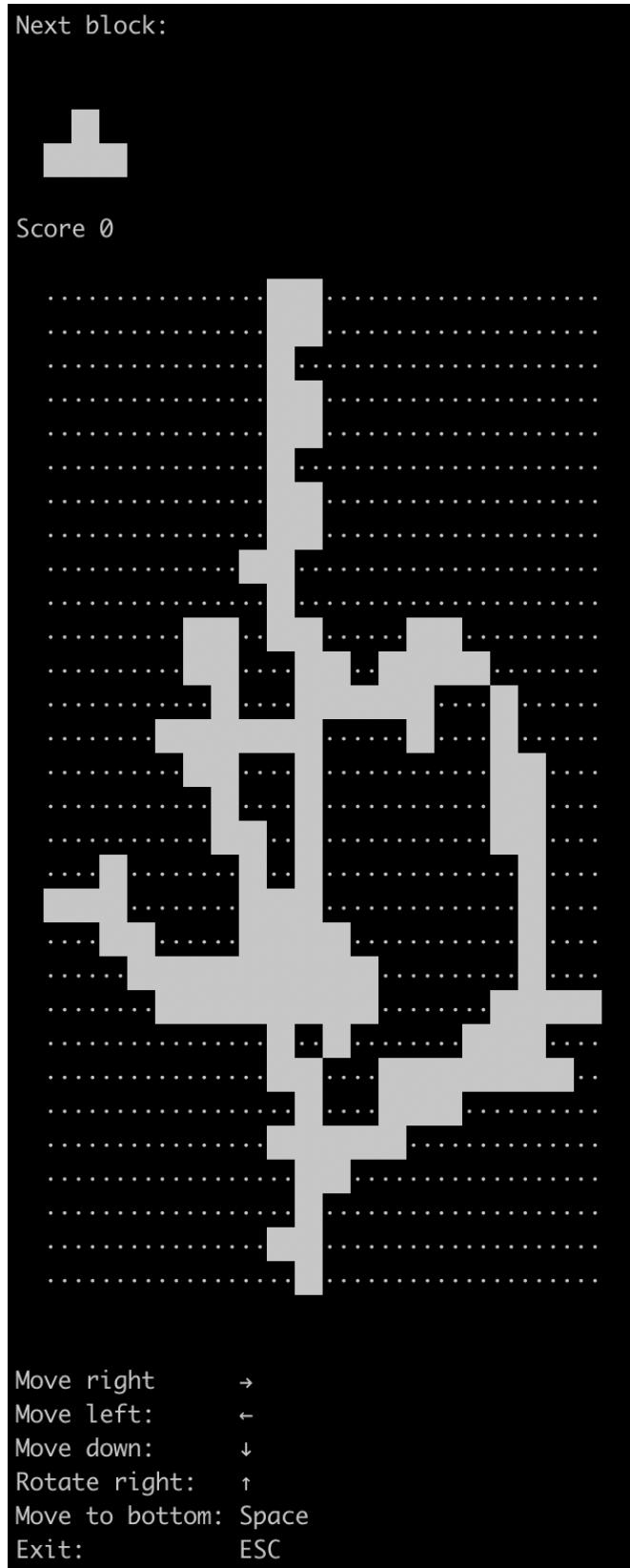
1.2 Ý tưởng

Trò chơi có bảy loại **khối hình**: I (thẳng đứng), J, L, O (vuông), S, T, Z.

Khi xoay các **khối hình** cơ bản này các góc tương ứng 90° , 180° , 270° thì ta sẽ được các **biến thể xoay** khác nhau của **khối hình**.

Game bắt đầu bằng một khoảng trống hình chữ nhật. Một loạt **khối hình** (và các **biến thể xoay** của nó) ngẫu nhiên rơi xuống. Người chơi sẽ xoay, di chuyển các **khối hình** để chúng rơi vào vị trí thích hợp. Mỗi khi người chơi lấp đầy 1 hàng ngang thì sẽ được điểm, đồng thời chương trình sẽ xóa hàng ngang đó ngay. Game sẽ kết thúc khi các **khối hình** bị xếp chồng đủ cao và chạm vào cạnh trên của khoảng trống. Các lệnh điều khiển gồm:

- Dịch trái, dịch phải
- Tăng tốc rơi
- Rơi xuống đáy ngay lập tức
- Xoay phải 90°



Hình 1.1: Tetris trên giao diện dòng lệnh (Console User Interface - CUI)

Phần 2

Ý tưởng giải thuật, các bài toán cốt lõi

Người ta thường lập trình Tetris chạy trên [giao diện đồ họa](#) (Graphical User Interface - GUI). Trong phạm vi tiểu luận này, tôi sẽ lập trình để chạy trên [giao diện dòng lệnh](#) (Console User Interface - CUI), do đó, tôi sẽ giới thiệu về một vài điểm đặc trưng của [giao diện dòng lệnh](#) này, cùng với một số yếu tố tương ứng trong giải thuật để chạy Tetris được trên đó.

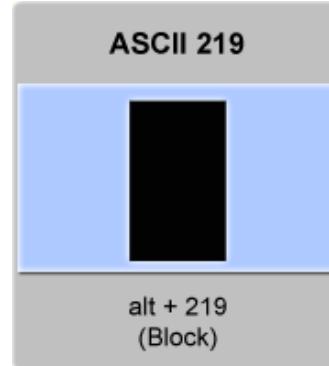
2.1 Về [giao diện dòng lệnh](#) (Console User Interface - CUI)

Đây là môi trường làm việc sơ khai đời đầu của máy tính điện tử. Trên màn hình xuất hiện 1 con trỏ nhấp nháy, báo hiệu sẵn sàng đợi lệnh từ người dùng. Người dùng ra lệnh bằng cách gõ một chuỗi [kí tự](#) (character) ASCII, kết thúc bằng 'enter'. Máy tính thực thi lệnh và xuất kết quả ra màn hình cũng ở dạng chuỗi [kí tự](#) trên một hoặc nhiều dòng. Con trỏ nhấp nháy xuất hiện trở lại, báo hiệu sẵn sàng đợi lệnh tiếp theo.

Câu hỏi: Làm sao để "in ra các [khối hình](#)" khi mà máy tính chỉ có thể xuất ra [kí tự](#) trong [giao diện dòng lệnh](#) này? Phần tiếp theo sẽ giải đáp.

2.2 Các kí tự đặc biệt trong giao diện dòng lệnh

Bên cạnh các **kí tự** chữ cái thông thường, bảng mã ascii còn chứa một loạt **kí tự** đặc biệt khác. Để có thể in ra các **khối hình**, ta sẽ nhanh trí sử dụng **kí tự** đặc biệt sau:



Hình 2.1: Block 219 on ASCII table

ASCII control characters		ASCII printable characters								Extended ASCII characters							
00	NUL (Null character)	32	space	64	@	96	`	128	ç	160	á	192	l	224	ó		
01	SOH (Start of Header)	33	!	65	A	97	a	129	ü	161	í	193	ł	225	þ		
02	STX (Start of Text)	34	"	66	B	98	b	130	é	162	ó	194	ł	226	ö		
03	ETX (End of Text)	35	#	67	C	99	c	131	â	163	ú	195	ł	227	ò		
04	EOT (End of Trans.)	36	\$	68	D	100	d	132	ä	164	ñ	196	—	228	ö		
05	ENQ (Enquiry)	37	%	69	E	101	e	133	à	165	ñ	197	ł	229	ö		
06	ACK (Acknowledgement)	38	&	70	F	102	f	134	á	166	ª	198	ã	230	µ		
07	BEL (Bell)	39	'	71	G	103	g	135	ç	167	º	199	Ã	231	þ		
08	BS (Backspace)	40	(72	H	104	h	136	ê	168	¸	200	ll	232	þ		
09	HT (Horizontal Tab)	41)	73	I	105	i	137	ë	169	®	201	Ł	233	ú		
10	LF (Line feed)	42	*	74	J	106	j	138	è	170	¬	202	Ł	234	ó		
11	VT (Vertical Tab)	43	+	75	K	107	k	139	í	171	½	203	Ł	235	ú		
12	FF (Form feed)	44	,	76	L	108	l	140	î	172	¼	204	Ł	236	ý		
13	CR (Carriage return)	45	-	77	M	109	m	141	í	173	í	205	=	237	ÿ		
14	SO (Shift Out)	46	.	78	N	110	n	142	Ã	174	«	206	‡	238	—		
15	SI (Shift In)	47	/	79	O	111	o	143	À	175	»	207	¤	239	‘		
16	DLE (Data link escape)	48	0	80	P	112	p	144	É	176	„	208	ð	240	≡		
17	DC1 (Device control 1)	49	1	81	Q	113	q	145	æ	177	„	209	đ	241	±		
18	DC2 (Device control 2)	50	2	82	R	114	r	146	Æ	178	„	210	Ê	242	≈		
19	DC3 (Device control 3)	51	3	83	S	115	s	147	ô	179	„	211	È	243	¾		
20	DC4 (Device control 4)	52	4	84	T	116	t	148	ö	180	„	212	È	244	¶		
21	NAK (Negative acknowl.)	53	5	85	U	117	u	149	ò	181	Á	213	í	245	§		
22	SYN (Synchronous idle)	54	6	86	V	118	v	150	ú	182	Á	214	í	246	÷		
23	ETB (End of trans. block)	55	7	87	W	119	w	151	ù	183	Á	215	í	247	°		
24	CAN (Cancel)	56	8	88	X	120	x	152	ÿ	184	©	216	í	248	°		
25	EM (End of medium)	57	9	89	Y	121	y	153	Ö	185	„	217	„	249	“		
26	SUB (Substitute)	58	:	90	Z	122	z	154	Ü	186	„	218	„	250	·		
27	ESC (Escape)	59	;	91	[123	{	155	ø	187	„	219	■	251	¹		
28	FS (File separator)	60	<	92	\	124		156	£	188	„	220	■	252	³		
29	GS (Group separator)	61	=	93]	125	}	157	Ø	189	¢	221	—	253	²		
30	RS (Record separator)	62	>	94	^	126	~	158	×	190	¥	222	—	254	■		
31	US (Unit separator)	63	?	95	—			159	f	191	„	223	■	255	nbsp		
127	DEL (Delete)																

Hình 2.2: ASCII table

Nếu ta xuất 2 **kí tự** liên tiếp nhau, thì sẽ được một hình chữ nhật kha khá giống hình vuông. Và đây chính là một **khối vuông cơ bản** mà ta cần.

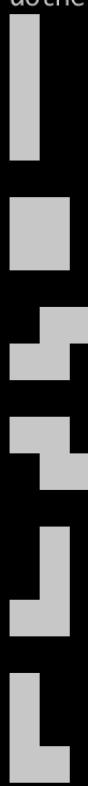
Nếu ta kết hợp với **kí tự** khoảng trắng (space), và in nhiều dòng **kí tự** thì sẽ được như hình 2.3

```
13:22:48 in BoxSync/learn-csharp/tetris on [?]main [+] via .NET 6.0.109 took 3s
→ dotnet run

```

Hình 2.3: In **khối hình** T trên giao diện dòng lệnh

Tương tự đối với các **khối hình** khác trên hình 2.4

```
13:26:52 in BoxSync/learn-csharp/tetris on [?]main [x+?] via .NET 6.0.109
→ dotnet run

```

Hình 2.4: In các **khối hình** còn lại trên giao diện dòng lệnh

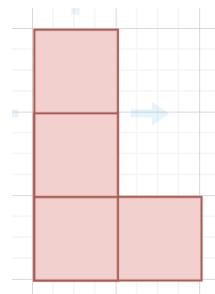
2.3 Xây dựng các khối hình và các biến thể xoay

Với cách tiếp cận như trên, ta thấy rằng mỗi **khối hình** chỉ đơn thuần là 1 chuỗi **kí tự** phù hợp. Nhanh trí, ta cũng nhận ra rằng mỗi **biến thể xoay** cũng sẽ tương ứng với 1 chuỗi **kí tự**. Tổng cộng ta sẽ cần 19 chuỗi là đủ để mô tả toàn bộ các **khối hình** và **biến thể xoay**.

Liệu ta có thể làm tốt hơn không?

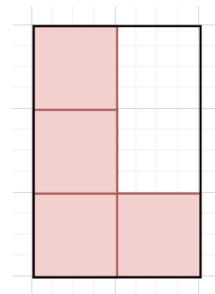
Mặc dù 19 chuỗi **kí tự** không phải nhiều, nhưng để thử thách hơn chút xíu, tôi xin giới thiệu giải thuật tạo ra chuỗi **kí tự** của **biến thể xoay** dựa trên chuỗi **kí tự** của **khối hình** gốc.

Lấy ví dụ với **khối hình L** (hình 2.5):



Hình 2.5: **Khối hình L**

Khối hình L đặt vừa vào trong một hình chữ nhật như hình 2.6:



Hình 2.6: **Khối hình L** và hình chữ nhật bao quanh

Khi đó, **khối hình L** sẽ tương ứng với ma trận L thế này:

$$L = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Phép chuyển vị ma trận L sẽ tạo ra ma trận L1 thế này:

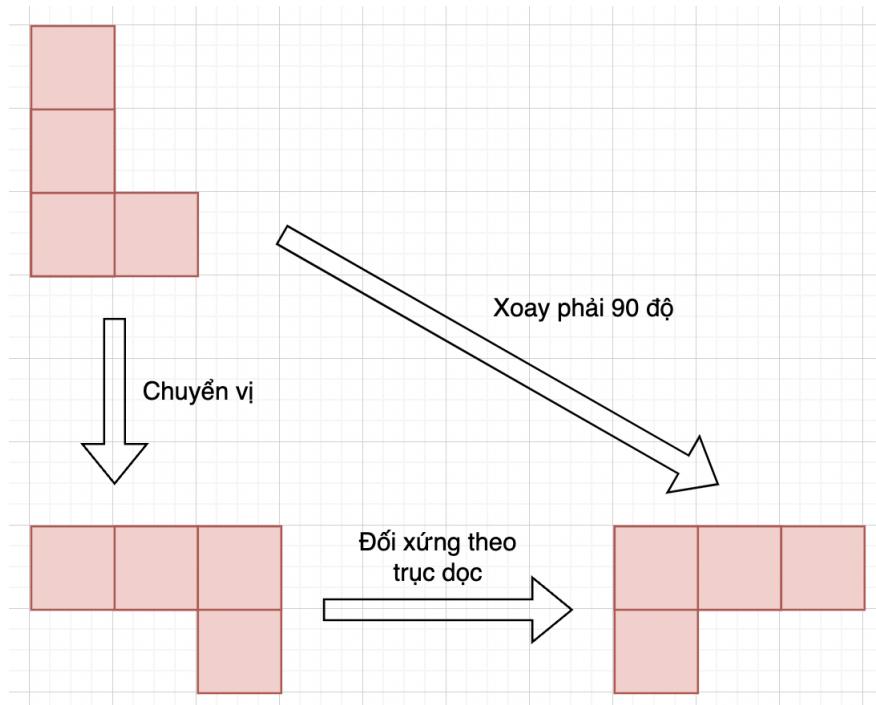
$$L1 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Tiếp theo, lấy đối xứng L1 qua trục dọc, ta được ma trận L2 như sau:

$$L2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Ma trận này tương ứng với **khối hình** L xoay phải 90° !

Tổng kết giải thuật, xem hình [2.7](#)



Hình 2.7: Giải thuật xoay phải 90°

Khối hình L xoay phải 180° hoặc 270° sẽ tương ứng với việc xoay phải 90° 2 lần hoặc 3 lần.

Doạn chương trình xoay phải 90° trong C# trông như sau:

Listing 2.1: Bắt sự kiện gõ phím

```
1 abstract class IBlock {
2     public Matrix Structure { get; private set; }
3     private void _transpose() {
4         Matrix newStructure = new Matrix();
5         newStructure.Empty(
6             this.Structure.Height(),
7             this.Structure.Width());
8
9         for (int row = 0; row < this.Structure.Height(); row++)
10            for (
11                int col = 0;
12                col < this.Structure.Width(); col++)
13            )
14                newStructure[col][row] =
15                    this.Structure[row][col];
16
17        this.Structure = newStructure;
18    }
19
20    private void _mirror() {
21        Matrix newStructure = new Matrix();
22        newStructure.Empty(
23            this.Structure.Width(),
24            this.Structure.Height());
25        for (int row = 0; row < this.Structure.Height(); row++)
26            for (
27                int col = 0;
28                col < this.Structure.Width(); col++)
29            {
30                int newCol = this.Structure.Width() - col - 1;
31                newStructure[row][newCol] =
32                    this.Structure[row][col];
33            }
34
35        this.Structure = newStructure;
36    }
37
38    public void RotateRight() {
39        lock (this) {
40            this._transpose();
41            this._mirror();
42        }
43    }
44}
```

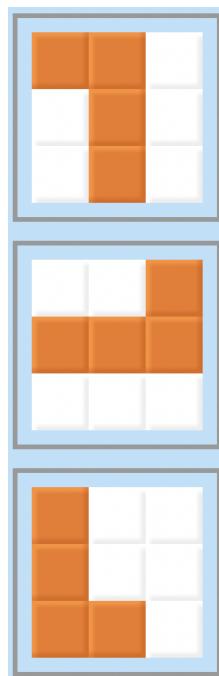
2.3.1 Các kiểu xoay

1. SRS
2. TGM
3. MTC
4. Atari

Xem thêm tại [Tet22]

Trong tiểu luận này, tôi sử dụng kiểu xoay Atari (hình 2.8), đặt cùng tên với nhà phát hành các loại máy chơi game mà thế hệ 8x và 9x đời đầu khá quen thuộc (đặc biệt là lại máy chơi game cầm tay thường bán trong cửa hàng tạp hóa - hình 2.9).

Kiểu xoay Atari này về cơ bản bỏ qua hiệu chỉnh tọa độ, luôn mặc định lấy điểm gốc trên-trái của khối chữ nhật tối thiểu bao quanh **khối hình** làm gốc. Ưu điểm: dễ thực thi giải thuật. Nhược điểm: Chuyển động của **khối hình** sẽ không được mượt, cũng như không thể thực hiện một số kĩ thuật chơi game nâng cao để chèn **khối hình** vào vị trí đẹp trong những tình huống phức tạp.



Hình 2.8: Kiểu xoay Atari



Hình 2.9: Máy chơi game cầm tay quen thuộc với thế hệ 8x - 9x

2.4 Tạo chuyển động trong môi trường dòng lệnh

Nguyên lý tạo ra chuyển động chính là xuất ra một loạt các khung hình đủ nhanh. Theo đó, ta chỉ cần:

- Xuất ra khung hình
- Xóa toàn màn hình giao diện dòng lệnh
- Xuất ra khung hình mới
- Xóa toàn màn hình giao diện dòng lệnh
- ... (cứ thế)

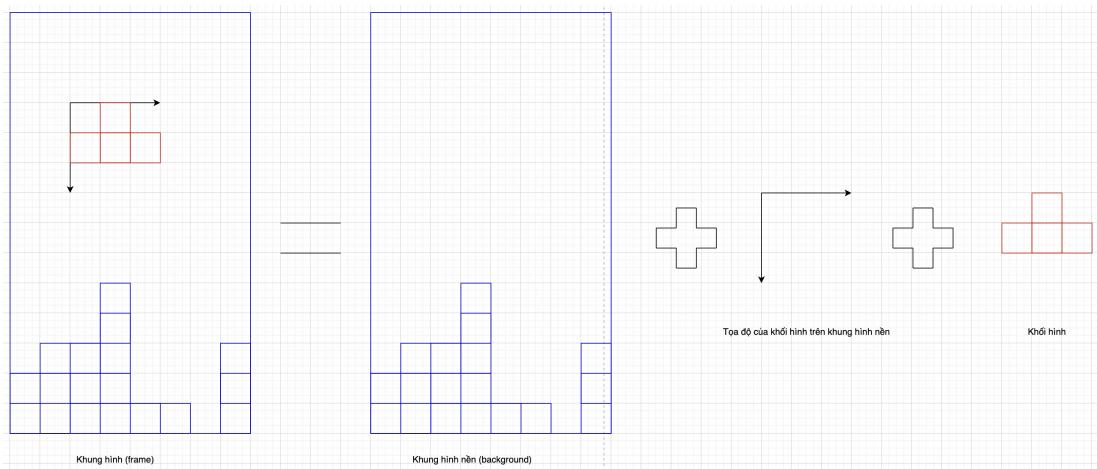
Xóa toàn màn hình giao diện dòng lệnh bằng lệnh `Console.Clear()`.

Khoảng không gian để các khối hình chuyển động ta gọi là `Screen`. Ta sẽ định nghĩa lớp `đối tượng` `Screen` này ở phần 3.3. Về cơ bản thì đây là một mảng 2 chiều, biểu diễn 1 lưới các khối vuông cơ bản.

Tập hợp các khối hình đã xếp chồng lên nhau (và do đó đứng yên) ta gọi là khung hình nền. Và khung hình nền này có kích thước bằng đúng mang 2

chiều định nghĩa trong lớp đối tượng Screen. Ta sẽ định nghĩa lớp đối tượng Background ở phần 3.2.

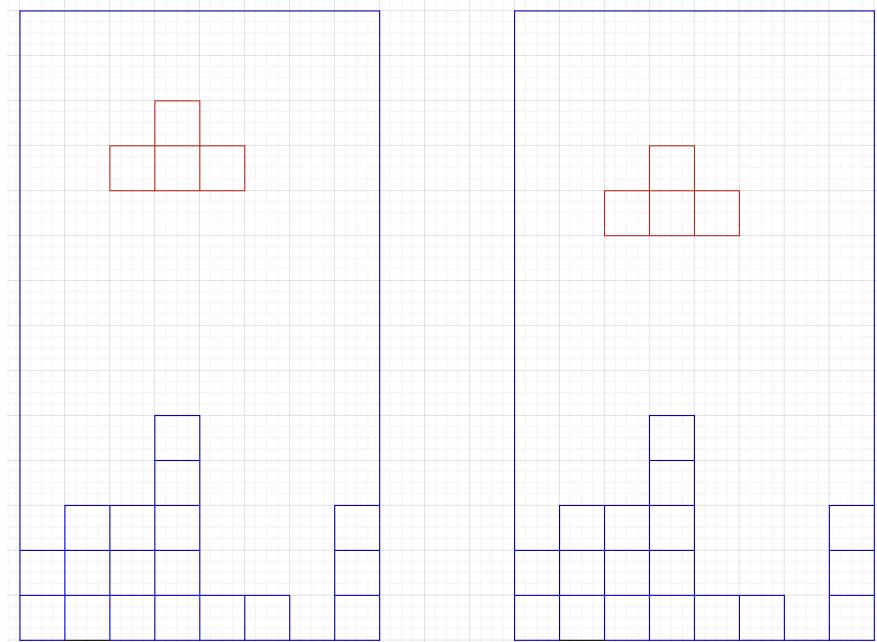
Một công thức đơn giản: Bằng cách chồng khối hình (Block) lên trên khung hình nền (Background), ta sẽ tạo ra một khung hình (Frame). Và ta định kì xuất khung hình này ra giao diện dòng lệnh thông qua một phương thức của Screen. Chi tiết sẽ đề cập ở phần 3.3



Hình 2.10: chồng khối hình lên trên khung hình nền tại một tọa độ cho trước

Với nguyên lý trên, mọi bài toán tạo chuyển động rời sẽ quy về bài toán tạo khung hình kế tiếp và bài toán chồng khối hình lên khung hình nền.

2.4.1 Chuyển động rơi



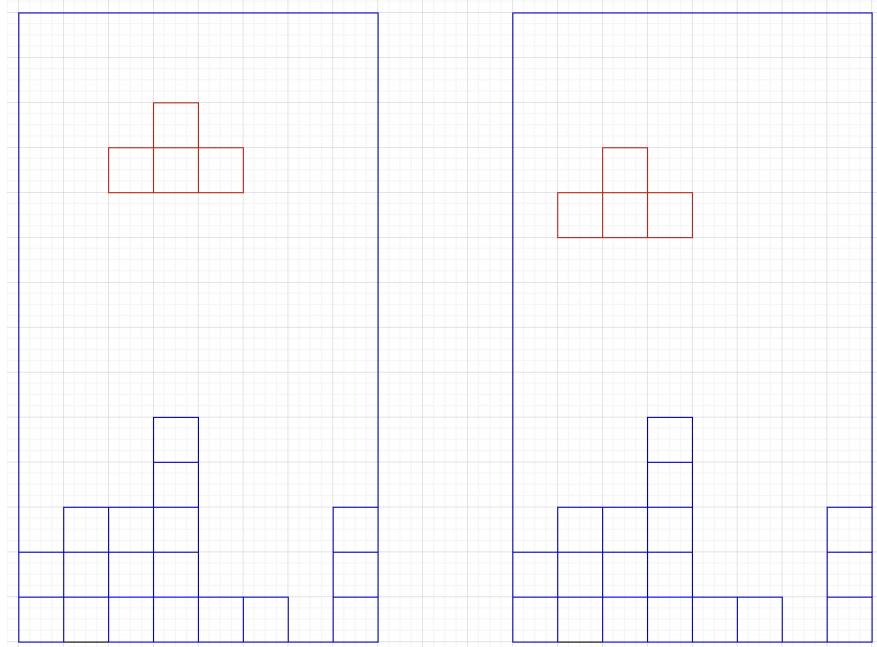
Hình 2.11: Chuyển động rơi

Chỉ cần tăng tọa độ dọc của [khối hình](#) lên 1 đơn vị trong quá trình xếp chồng lên [khung hình nền](#). Dĩ nhiên ta sẽ cần kiểm tra các điều kiện [chồng lán](#) (phần 2.7.1) và chạm mép (phần 2.7.2). Xem hình 2.11.

Listing 2.2: Giải thuật rơi xuống 1 đơn vị

```
1 namespace Tetriz {
2     abstract class IBlock {
3         public int OffsetX { get; private set; } = 0;
4         public int OffsetY { get; private set; } = 0;
5         public void MoveDown() {
6             lock (this) {
7                 this.OffsetY += 1;
8             }
9         }
10     ...
}
```

2.4.2 Dịch trái - phải



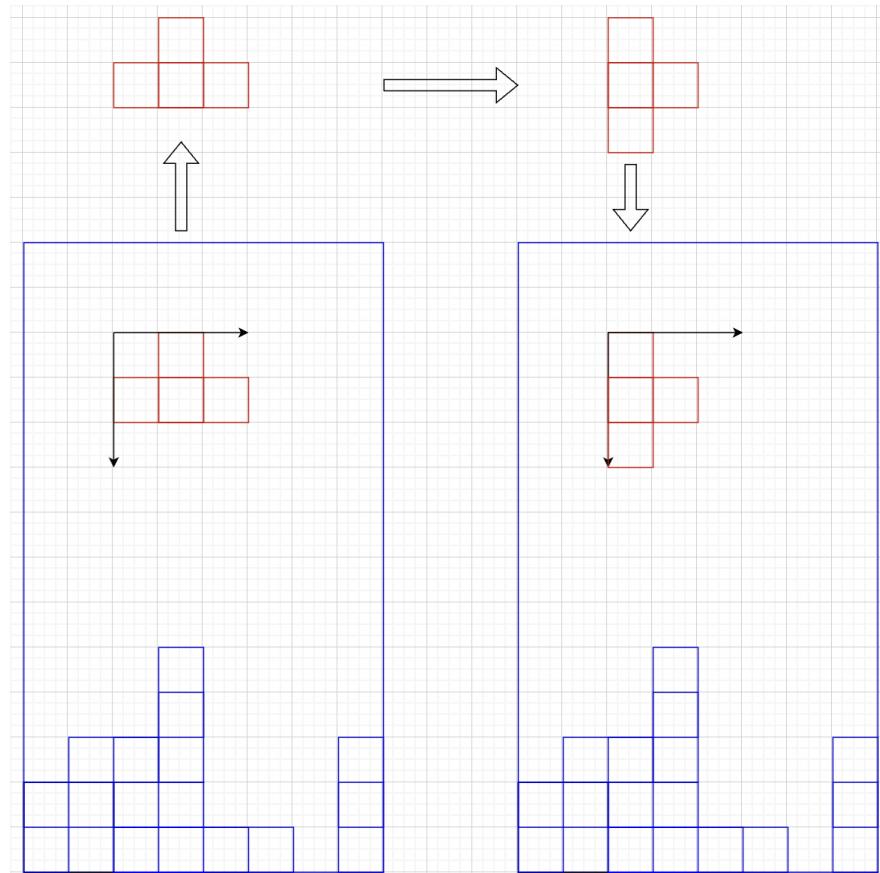
Hình 2.12: Dịch trái

Listing 2.3: Giải thuật dịch trái - phải 1 đơn vị

```
1  namespace Tetriz {
2      abstract class IBlock {
3          public int OffsetX { get; private set; } = 0;
4          public int OffsetY { get; private set; } = 0;
5          public void MoveLeft() {
6              lock (this) {
7                  this.OffsetX -= 1;
8              }
9          }
10         public void MoveRight() {
11             lock (this) {
12                 this.OffsetX += 1;
13             }
14         }
15     ...
}
```

Tăng giảm tọa độ ngang của **khối hình** 1 đơn vị trong quá trình xếp chồng lên **khung hình nền**. Cần kiểm tra điều kiện **chồng lấn** (phần 2.7.1) và chạm mép (phần 2.7.2). Xem hình 2.12

2.4.3 Xoay



Hình 2.13: Bóc tách [khối hình](#), xoay và gán lại vào đúng tọa độ cũ

Tiến hành xoay [khối hình](#) một góc 90° để tạo ra [biến thể xoay](#) theo giải thuật ở phần 2.3, sau đó xếp chồng lên [khung hình nền](#). Cần kiểm tra điều kiện [chồng lần](#) (phần 2.7.1) và chạm mép (phần 2.7.2). Xem hình 2.13

2.5 Bắt và xử lý sự kiện gõ phím

Listing 2.4: Bắt sự kiện gõ phím

```
1 private static void _readKeysThread()
2 {
3     Boolean loop = true;
4     while (loop)
5     {
6         ConsoleKey keyPressed = Console.ReadKey(true).Key;
7         switch (keyPressed)
8         {
9             case ConsoleKey.Escape:
10                 loop = false;
11                 break;
12             default:
13                 screen.HandleKey(keyPressed);
14                 break;
15         }
16     }
17     ...
18 }
```

Thư viện chuẩn của C# có hỗ trợ bắt sự kiện gõ phím bằng `Console.ReadKey(true)`, `true` ở đây tức là không cho hiển thị ký tự của phím vừa gõ lên giao diện dòng lệnh. Bạn tham khảo thêm ở [Mic22a].

Ta xử lý sự kiện này bằng cách gọi phương thức `HandleKey` của thực thể `screen` (sẽ đề cập ở phần 3.3)

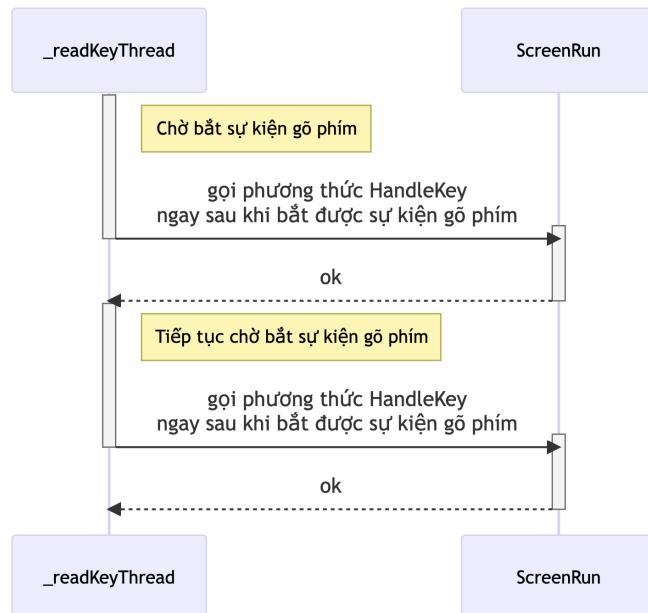
2.6 Xử lý song song theo luồng (thread)

Như đã biết ở phần 2.4, ta sử dụng vòng lặp vô tận để tạo các hiệu ứng chuyển động. Vậy ta sẽ đón bắt lấy sự kiện gõ phím ở đâu và lúc nào?

Có 2 giải pháp khả dĩ:

1. Đón bắt lấy sự kiện ở đầu, cuối hoặc một chỗ nào đó trong từng vòng lặp. Với cách này, ta sẽ có 1 khe thời gian mà ở đó chương trình đang bận bịu tính toán các logic khác mà không thể lắng nghe sự kiện gõ phím, tức là có khả năng bỏ sót sự kiện đó. Dù khe thời gian này khá hẹp, nhưng điều này có thể tạo ra trải nghiệm tệ khi chơi.

2. Dồn bắt lấy **sự kiện** ở một tiến trình logic độc lập, chạy song song với vòng lặp vô tận hiện tại. Cơ chế này gọi là **đa luồng** (multi-thread). C# cũng như rất nhiều ngôn ngữ hiện đại đề hỗ trợ cơ chế lập trình kiểu này. Phương án này giúp loại bỏ hoàn toàn lo ngại bỏ lỡ **sự kiện** như trên.



Hình 2.14: Cơ chế **đa luồng** xử lí song song

Đọc thêm về cách sử dụng thư viện `Thread` tại [Mic22b].

Listing 2.5: Xử lí song song bằng cơ chế **đa luồng**

```
1  namespace Tetriz
2  {
3      class Program
4      {
5          private static Screen screen;
6          private static void _readKeysThread()
7          {
8              Boolean loop = true;
9              while (loop)
10             {
11                 ConsoleKey keyPressed = Console.ReadKey(true).Key;
12                 switch (keyPressed)
13                 {
14                     case ConsoleKey.Escape:
15                         loop = false;
16                         break;
17                     default:
18                         screen.HandleKey(keyPressed);
19                         break;
20                 }
21             }
22             Console.Clear();
23             Console.WriteLine(Const.TextGameOver);
24             System.Environment.Exit(0);
25         }
26         private static void _initKeyReading()
27         {
28             Thread keyReading = new Thread(_readKeysThread);
29             keyReading.IsBackground = true;
30             keyReading.Start();
31         }
32         static void Main()
33         {
34             _initKeyReading();
35             screen = new Screen(
36                 Const.DefaultScreenWidth,
37                 Const.DefaultScreenHeight);
38             screen.Run();
39         }
40     }
41 }
```

Nguyên lí:

Trong **phương thức** Main, ta gọi **phương thức** _initKeyReading. Trong **phương thức** _initKeyReading, ta tiến hành khởi tạo **luồng** mới gắn với **phương thức** _readKeysThread và kích hoạt **luồng** này chạy ngầm, xem giải thuật ??

Listing 2.6: Kích hoạt **đa luồng** chạy ngầm

```
1 ...  
2 Thread keyReading = new Thread(_readKeysThread);  
3 keyReading.IsBackground = true;  
4 keyReading.Start();  
5 ...
```

2.7 Sinh khối hình ngẫu nhiên kế tiếp

Listing 2.7: Kích hoạt **đa luồng** chạy ngầm

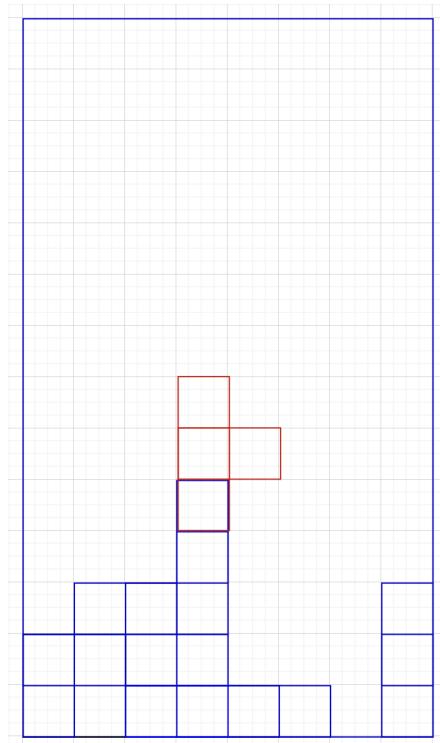
```
1 namespace Tetriz {
2     class Screen {
3         ...
4         private List<IBlock> _blockList = new List<IBlock>() {
5             new BlockI(),
6             new BlockJ(),
7             new BlockL(),
8             new BlockO(),
9             new BlockS(),
10            new BlockT(),
11            new BlockZ(),
12        };
13        private IBlock _randomBlock() {
14            Random rnd = new Random();
15            int blockIndex = rnd.Next(0, this._blockList.Count);
16            Console.WriteLine("{0} / {1}",
17                blockIndex, this._blockList.Count);
18            IBlock newBlock = this._blockList.ElementAt(blockIndex);
19            int rotateCount = rnd.Next(0, 4);
20            for (int i = 0; i < rotateCount; i++)
21                newBlock.RotateRight();
22            return newBlock;
23        }
24        ...
}
```

Ý tưởng:

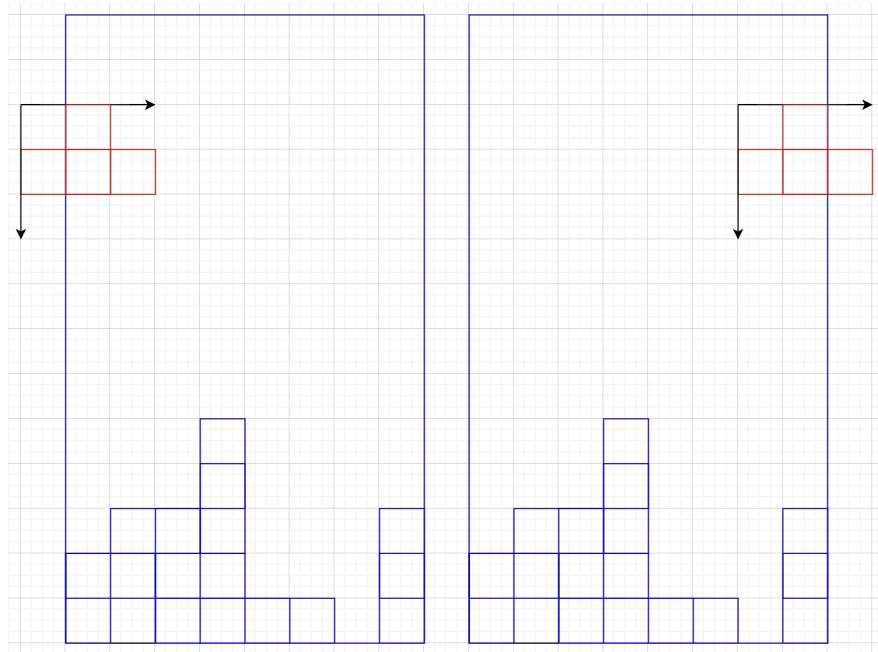
- Khởi tạo sẵn một danh sách các **khối hình** khả dĩ
- Sử dụng thư viện Random để sinh ra chỉ số ngẫu nhiên để chọn **khối hình** ngẫu nhiên.
- Tiếp tục sinh ra chỉ số ngẫu nhiên thứ 2 để chọn **biến thể xoay** ngẫu nhiên của **khối hình** kể trên.

2.7.1 Kiểm tra chồng lấn

Trong giải thuật chồng khối hình lên trên khung hình nền, ta cần kiểm tra xem liệu có khối vuông cơ bản nào trong cấu thành của khối hình bị trùng với khối vuông cơ bản sẵn có trên khung hình nền hay không. Giải thuật này nhìn chung đơn giản, chỉ cần canh theo tọa độ của khối hình, sau đó chạy vòng lặp kiểm tra đối sánh.



Hình 2.15: Chồng lấn lên các khối hình đã xếp



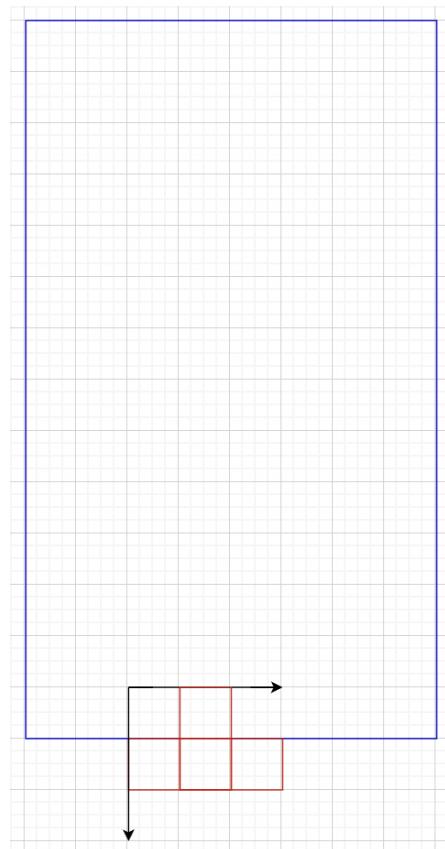
Hình 2.16: Vi phạm mép trái, mép phải

Listing 2.8: Phương thức kiểm tra chồng lân

```

1  namespace Tetriz {
2      class Background {
3          private Matrix _matrix;
4          ...
5          public Boolean HasCollision(IBlock block) {
6              Matrix bMatrix = block.Structure;
7              for (int row = 0; row < bMatrix.Height(); row++) {
8                  for (int col = 0; col < bMatrix.Width(); col++) {
9                      if (bMatrix[row][col] != Const.X)
10                          continue;
11                      int pRow = (int)(block.OffsetY + row);
12                      int pCol = (int)(block.OffsetX + col);
13                      if (IsValidPixel(pRow, pCol))
14                          if (this._matrix[pRow][pCol] == Const.X)
15                              return true;
16                  }
17              }
18              return false;
19          }
20          ...

```



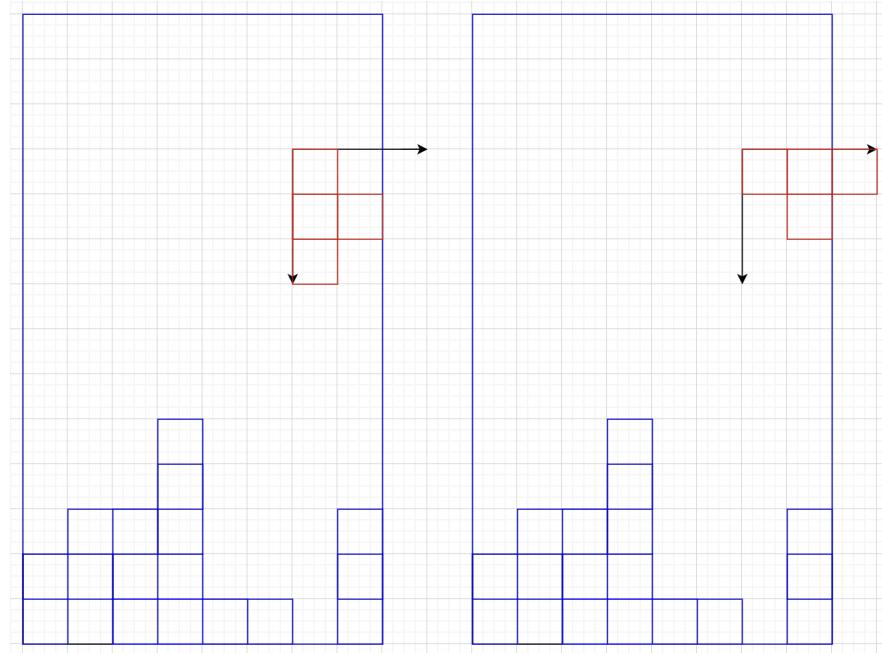
Hình 2.17: Vi phạm mép dưới

2.7.2 Kiểm tra mép trái, mép phải, mép dưới

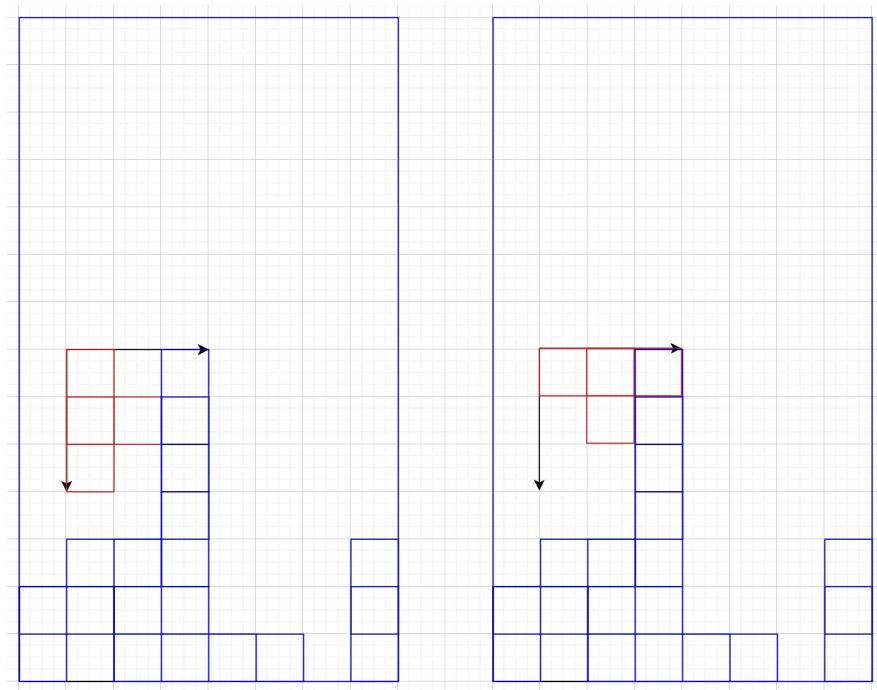
Listing 2.9: Phương thức kiểm tra vi phạm mép trái/phải/dưới

```
1 namespace Tetriz {
2     class Background {
3         ...
4         private Matrix _matrix;
5         public Boolean IsOutOfMargin(IBlock block) {
6             return
7                 IsOutOfMarginBottom(block) ||
8                 IsOutOfMarginLeft(block) ||
9                 IsOutOfMarginRight(block);
10            }
11
12        public Boolean IsOutOfMarginLeft(IBlock block) {
13            return block.OffsetX < 0;
14        }
15
16        public Boolean IsOutOfMarginRight(IBlock block) {
17            long maxRight = block.OffsetX + block.Structure.Width();
18            return maxRight > this._matrix.Width();
19        }
20
21        public Boolean IsOutOfMarginBottom(IBlock block) {
22            long maxBottom = block.OffsetY + block.Structure.Height
23            ();
24            return maxBottom > this._matrix.Height();
25        }
26        ...
27    }
```

2.7.3 Kiểm tra khả năng xoay khi gần mép

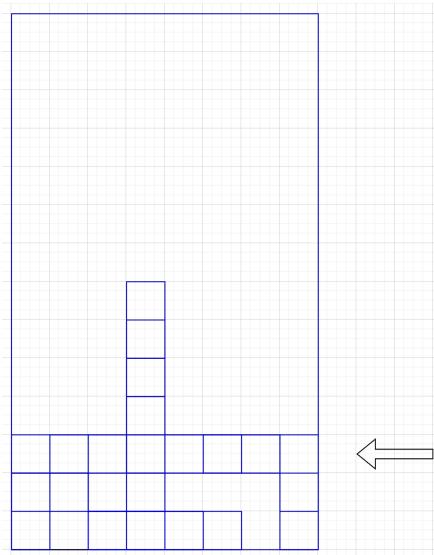


Hình 2.18: Vi phạm mép phải sau khi xoay



Hình 2.19: Chồng lán mép phải sau khi xoay

2.7.4 Kiểm tra hàng đầy, tính điểm và xóa hàng đầy



Hình 2.20: Hàng đầy

Listing 2.10: Phương thức kiểm tra hàng đầy

```

1 namespace Tetriz {
2     class Background {
3         ...
4         private Matrix _matrix;
5
6         private Boolean IsFilledRow(List<String> row) {
7             foreach (var pixel in row)
8                 if (pixel == Const.Dot)
9                     return false;
10            return true;
11        }
12
13        public int CountFilledRows() {
14            int count = 0;
15            foreach (var row in this._matrix)
16                if (IsFilledRow(row))
17                    count += 1;
18            return count;
19        }
20        ...

```

Duyệt qua từng hàng của khung hình nền, so sánh từng khối vuông cơ bản xem nó có trống không. Giải thuật này vô cùng đơn giản.

Listing 2.11: Phương thức kiểm tra chồng lán, label=algo:collision

```
1 namespace Tetriz {
2     class Background {
3         private Matrix _matrix;
4         ...
5         private void TranslateRowIToK(int i, int k) {
6             for (int x = 0; x < this._matrix.Width(); x++)
7             {
8                 this._matrix[k][x] = this._matrix[i][x];
9                 this._matrix[i][x] = Const.Dot;
10            }
11        }
12
13        private void TranslationRangeDown(int targetRowIndex) {
14            if (targetRowIndex < 0)
15                return;
16            if (targetRowIndex >= this._matrix.Height())
17                return;
18            for (int i = targetRowIndex - 1; i > 0; i--)
19                TranslateRowIToK(i, i + 1);
20            for (int i = 0; i < this._matrix.Width(); i++)
21                this._matrix[0][i] = Const.Dot;
22            return;
23        }
24
25        public void EraseFilledRows() {
26            // check filled row from top to bottom
27            for (int i = 0; i < this._matrix.Height(); i++) {
28                var row = this._matrix[i];
29                if (IsFilledRow(row))
30                    TranslationRangeDown(i);
31            }
32        }
33        ...
34    }
```

Phương thức `TranslateRowIToK` sẽ sao chép hàng thứ i (tính từ trên xuống) vào hàng thứ k . Hàng thứ i vẫn giữ nguyên như cũ.

Phương thức `TranslationRangeDown` sử dụng phương thức `TranslateRowIToK` để tịnh tuyến toàn bộ các hàng ở trên hàng `targetRowIndex` đi xuống 1 hàng. Và hàng trên cùng sẽ bị xóa trắng. Kết quả của quá trình này tương đương với việc xóa một hàng đầy nào đó, xem mã nguồn của phương thức `EraseFilledRows` ở ??

Phần 3

Tổng quan cấu trúc chương trình

3.1 Khai báo lớp đối tượng Block

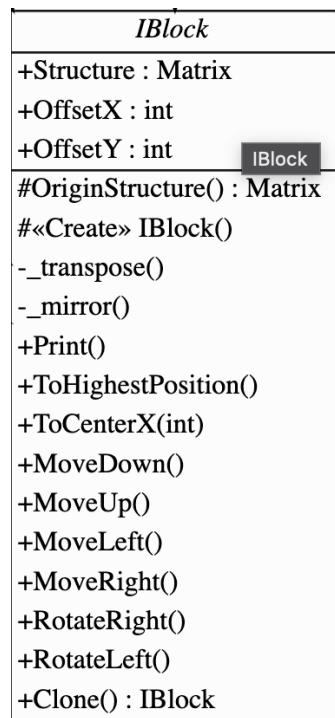
Thay vì lặp đi lặp lại các **phương thức** trên tất cả các khai báo **khối hình**, ta sẽ định nghĩa một **lớp đối tượng trừu tượng** `IBlock` chứa đủ các **phương thức** chung đó. Dựa trên đó, các **khối hình** khác sẽ kế thừa **lớp đối tượng trừu tượng** `IBlock` này. Xem đoạn [mã nguồn 3.1](#) và [hình 3.1](#).

Listing 3.1: IBlock

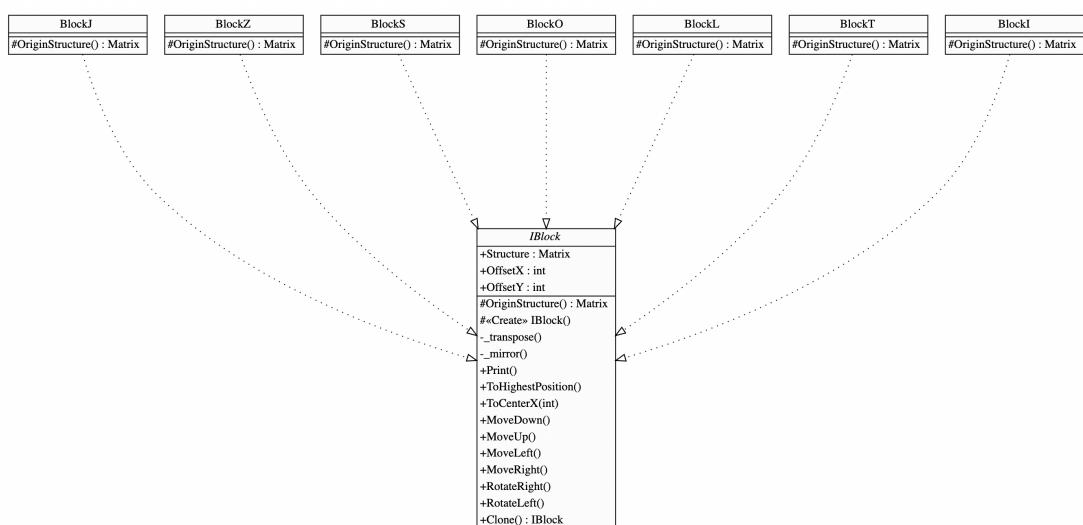
```
1 namespace Tetriz{
2     abstract class IBlock {
3         protected abstract Matrix OriginStructure();
4         public Matrix Structure { get; private set; }
5         public int OffsetX { get; private set; } = 0;
6         public int OffsetY { get; private set; } = 0;
7
8         protected IBlock() {
9             if (this.Structure == null)
10                 this.Structure = this.OriginStructure();
11         }
12     ...
13 }
```

Có 1 **phương thức** trừu tượng mà các **lớp đối tượng** con cần phải **thực thi**: `OriginStructure`. **Phương thức** sẽ được gọi ngay khi khởi tạo để có được một `Matrix` tương ứng với từng **khối hình** cụ thể gán vào **thuộc tính** `Structure`

`OffsetX` và `OffsetY` tương ứng là 2 **thuộc tính** đánh dấu tọa độ của **khối hình** theo phương ngang (từ trái sang phải) và phương đứng (từ trên xuống dưới) trên **khung hình nền** (xem phần [3.2](#)).



Hình 3.1: IBlock



Hình 3.2: Lớp đối tượng trừu tượng IBlock và các lớp đối tượng kế thừa

Các **phương thức** MoveDown, MoveUp, MoveLeft, MoveRight tác động lên **thuộc tính** tọa độ OffsetX và OffsetY để tạo dịch chuyển theo phương ngang và thẳng đứng. Các phương thức RotateRight, RotateLeft tác động lên **thuộc tính** Structure để tạo ra chuyển động xoay. 2 **phương thức** private _transpose, _mirror sử dụng trong giải thuật tạo **bien thể xoay** như đã đề cập ở phần 2.3

2 **phương thức** ToHighestPosition, ToCenterX sử dụng để thiết lập tọa độ khởi đầu cho mỗi **khối hình** ngẫu nhiên kế tiếp.

Phương thức Clone giúp tạo **thực thể** tương tự, cùng **lớp đối tượng** với **thực thể** gốc. Đây là một **phương thức** tiện ích sẽ sử dụng nhiều trong giải thuật chồng **khối hình** lên **khung hình nền** ở phần 3.2 và 3.3

Ví dụ: Ứng với **khối hình** T, ta định nghĩa lớp **đối tượng** kế thừa **lớp đối tượng** **trừu tượng** IBlock như hình 3.3

Listing 3.2: Block T

```

1 namespace Tetriz {
2     class BlockT : IBlock {
3         protected override Matrix OriginStructure()
4             => new Matrix() {
5                 new List<String>() {Const._, Const.X, Const._},
6                 new List<String>() {Const.X, Const.X, Const.X},
7             };
8         }
9 }
```

Trong đó, Const._ và Const.X là 2 hằng số định nghĩa trong **lớp đối tượng** Const, ứng với **kí tự** khoảng trắng và **kí tự** ASCII 219 như đã đề cập ở phần 2.2. Xem hình 3.3

```

1  namespace Tetriz
2  {
3      public static class Const
4      {
5          public const int DefaultScreenWidth = 20;
6          public const int DefaultScreenHeight = 30;
7          public const int FrameDelay = 600; // 1000 millisecond
8          public static string X = "█";
9          public static string _ = " ";

```

Hình 3.3: Định nghĩa **khối vuông** cơ bản

Background
-_matrix : Matrix
-InitMatrix(uint, uint) : Matrix
-IsValidPixel(int, int) : Boolean
-IsFilledRow(List<String>) : Boolean
-TranslateRowIToK(int, int)
-TranslationRangeDown(int)
+«Create» Background(uint, uint)
+TryToMergeBlock(IBlock) : Matrix
+MergeBlock(IBlock)
+CanMoveDown(IBlock) : Boolean
+IsOutOfMargin(IBlock) : Boolean
+IsOutOfMarginLeft(IBlock) : Boolean
+IsOutOfMarginRight(IBlock) : Boolean
+IsOutOfMarginBottom(IBlock) : Boolean
+HasCollision(IBlock) : Boolean
+CountFilledRows() : int
+EraseFilledRows()
+Show()

Hình 3.4: Background

Matrix
+Height() : int
+Width() : int
+Empty(int, int)
+Clone() : Matrix
+Print()

Hình 3.5: Matrix

3.2 Khai báo lớp đối tượng Background

Xem hình 3.5, chú ý lớp đối tượng **Matrix** (hình 3.5) sử dụng trong định nghĩa lớp đối tượng **Background**

Thuộc tính `_matrix` lưu trữ **thực thể** của lớp đối tượng **Matrix**. Mục đích là lưu trữ một mảng 2 chiều biểu diễn **khung hình nền** thực tế trên game. Thực chất lớp đối tượng **Matrix** ở đây chỉ là một biệt hiệu của kiểu `List<List<String>`, đính kèm theo một số **phương thức** để quá trình viết **mã nguồn** trở nên dễ dàng hơn, gọn hơn.

Phương thức `MergeBlock` cố gắng chèn khôi hình `IBlock` lên khung hình nền bằng cách gộp (merge) ma trận `IBlock.Structure` với `Background._matrix`. **Phương thức** `IsValidPixel` thực ra chỉ kiểm tra xem 1 tọa độ có thực sự hợp lệ trên khung hình nền hay không, và ta cần phương thức trong quá trình gộp

ma trận vừa đề cập.

[Phương thức](#) TryToMergeBlock sẽ thử gộp và kiểm tra các điều kiện chồng lấn, vi phạm mép trái/phải/dưới. Phương thức này sẽ gọi lại [phương thức](#) MergeBLock.

Các [phương thức](#) IsOutOfMargin, IsOutOfMarginBottom, IsOutOfMarginLeft, IsOutOfMarginRight đã đề cập ở phần [2.7.2](#).

[Phương thức](#) IsFilledRow, TranslateRowIToK, TranslationRangeDown, CountFilledRows, EraseFilledRows đã đề cập ở phần [2.7.4](#).

[Phương thức](#) Show thực ra chỉ nhầm in ra kết quả trực quan, để [gõ lỗi](#) (debug) trong quá trình viết nên chương trình này mà thôi.

Screen	
-_score : int	
-_background : Background	
-_currentBlock : IBlock	
-_nextBlock : IBlock	
-_keyPressed : ConsoleKey	
-_width : int	
-_height : int	
-_blockList : List<IBlock>	
-_randomBlock() : IBlock	
-_handleKey(ConsoleKey)	
-_draw()	
-_prepareForNextTurn()	
-_scoring()	
-_gameOver()	
+«Create» Screen(int, int)	
+HandleKey(ConsoleKey)	
+MoveDownAndCheck() : Boolean	
+Run()	

Hình 3.6: Screen

3.3 Khai báo lớp đối tượng Screen

Lớp đối tượng này quản lí các **thuộc tính** liên quan đến phần giao diện game, những cái mà người chơi sẽ thực sự nhìn thấy trên **giao diện dòng lệnh**.

Cụ thể:

Thuộc tính `_width`, `_height` lưu trữ kích thước màn hình game, cũng là kích thước của **khung hình nền**.

Thuộc tính `_currentBlock`, `_nextBlock` lần lượt lưu trữ **khối hình** hiện tại (là **khối hình** đang rơi, đang xoay, đang cần sắp xếp) và **khối hình** ngẫu nhiên sẽ xuất hiện sau khi sắp xếp xong **khối hình** hiện tại.

Thuộc tính `_keyPressed` lưu trữ phím đã gõ gần nhất. Mục đích chính là để **gõ lỗi** (debug) trong quá trình phát triển chương trình.

Thuộc tính `_score` lưu trữ điểm số. Cứ mỗi 1 hàng được lấp đầy thì chương trình sẽ xóa hàng đó và cộng thêm 1 điểm.

Thuộc tính `_background` gắn với **thực thể** của **lớp đối tượng** `Background`.

Thuộc tính `_blockList` lưu trữ mảng tất cả các **khối hình** cơ bản, sử dụng trong giải thuật tạo **khối hình** ngẫu nhiên kế tiếp như đã đề cập ở phần 2.7.

Thực thể `screen` của **lớp đối tượng** `Screen` sẽ quản lí các **thực thể** của **lớp đối**

tương Background và IBlock. Nó sẽ sinh, hủy, thao tác trên các **thực thể** được gán trong **thuộc tính** này trongn các **phương thức**:

- `_randomBLock`: đã đề cập ở phần [2.7](#)
- `HandlerKey`: nhận vào đối số `ConsoleKey`, tùy theo phím đã gõ mà sẽ gọi các **phương thức** dịch chuyển hoặc xoay **khối hình** hiện tại `_currentBLock` thông qua một. **Phương thức** này sẽ cẩn trọng khóa cứng chương trình (lock) và mở khóa sau khi chạy xong để tránh hiện tượng **tương tranh** (race-condition) thường thấy trong các xử lí **đa luồng** hoặc các **cơ chế hướng sự kiện**.
- `_draw`: vẽ một **khung hình** (frame) lên **giao diện dòng lệnh**
- `_scoring`: khá dễ đoán, đúng như tên gọi, nó sẽ kiểm tra hàng đầy, đếm và cộng điểm.
- `_prepareForNextTurn`: sẽ được gọi ngay sau khi **khối hình** hiện tại `_currentBLock` hoàn tất sắp xếp.
- `Run`: kích hoạt vòng lặp vô tận, tạo **khung hình** mới sau mỗi khoảng thời gian chờ cố định, định nghĩa trong `Const.FrameDelay`

Toàn văn **mã nguồn** của chương trình này có thể xem tại [\[Tha22\]](#)

Chú giải thuật ngữ

đa luồng (multi-thread). [16](#), [17](#), [18](#), [19](#), [34](#)

biến thể xoay (rotation variation). [ii](#), [1](#), [6](#), [14](#), [19](#), [30](#)

chồng lấn (collision). [ii](#), [12](#), [13](#), [14](#), [20](#), [21](#), [25](#), [27](#)

cơ chế hướng sự kiện (event-orientated mechanism). [34](#)

giao diện đồ họa (Graphical User Interface - GUI). [3](#)

giao diện dòng lệnh (Console User Interface - CUI). , [i](#), [ii](#), [2](#), [3](#), [4](#), [5](#), [10](#), [11](#), [15](#), [33](#), [34](#)

gõ lỗi (debug). [32](#), [33](#)

khoảng trống (space). [5](#)

khung hình nền (background-frame). [10](#), [11](#), [12](#), [13](#), [14](#), [20](#), [26](#), [28](#), [30](#), [31](#), [33](#)

khung hình (frame). [10](#), [11](#), [34](#)

khối vuông cơ bản (basic block). [1](#), [4](#), [10](#), [20](#), [26](#), [30](#)

khối hình (block). [ii](#), [1](#), [3](#), [4](#), [5](#), [6](#), [7](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [19](#), [20](#), [28](#), [30](#), [31](#), [33](#), [34](#)

kí tự (character). [ii](#), [3](#), [4](#), [5](#), [6](#), [30](#)

luồng (thread). [ii](#), [15](#), [18](#)

lớp đối tượng (class). [ii](#), [10](#), [11](#), [28](#), [29](#), [30](#), [31](#), [33](#)

mã nguồn (source-code). [27](#), [28](#), [31](#), [34](#)

phương thức (method). [11](#), [15](#), [18](#), [21](#), [23](#), [26](#), [27](#), [28](#), [30](#), [31](#), [32](#), [34](#)

sự kiện (event). [ii](#), [8](#), [15](#), [16](#)

thuộc tính (attribute). [28](#), [30](#), [31](#), [33](#), [34](#)

thực thi (implement). [28](#)

thực thể (instance). [15](#), [30](#), [31](#), [33](#), [34](#)

trùu tượng (abstract). [28](#), [29](#), [30](#)

tương tranh (race-condition). [34](#)

Tham khảo

- [Mic22a] Microsoft. Console.readkey method, 2022.
<https://learn.microsoft.com/en-us/dotnet/api/system.console.readkey?view=net-6.0>. 15
- [Mic22b] Microsoft. Thread, 2022. <https://learn.microsoft.com/en-us/dotnet/api/system.threading.thread?view=net-6.0>. 16
- [Tet22] Tetris. Các kiểu xoay tetris, 2022. <https://tetris.fandom.com/wiki/SRS>. 9
- [Tha22] Tran Duy Thanh. Mã nguồn chương trình sử dụng trong tiểu luận, 2022. <https://github.com/tranduythanh/learn-csharp/tree/main/tetris>. 34